Data Structures and Algorithms for Bioinformatics

Philip Machanick

DATA STRUCTURES AND ALGORITHMS FOR BIOINFORMATICS
First draft edition, 2017; updates 2019

Author:	Machanick, Philip, 1957-
Title:	Data structures and algorithms for bioinformatics / Philip Machanick
Edition:	draft ed.
Publisher:	Grahamstown, South Africa : RAMpage Research, 2017, 2019.
ISBN:	xxx-x-xxxxxxxx-x-x (pbk.)
LoC classification :	QA76

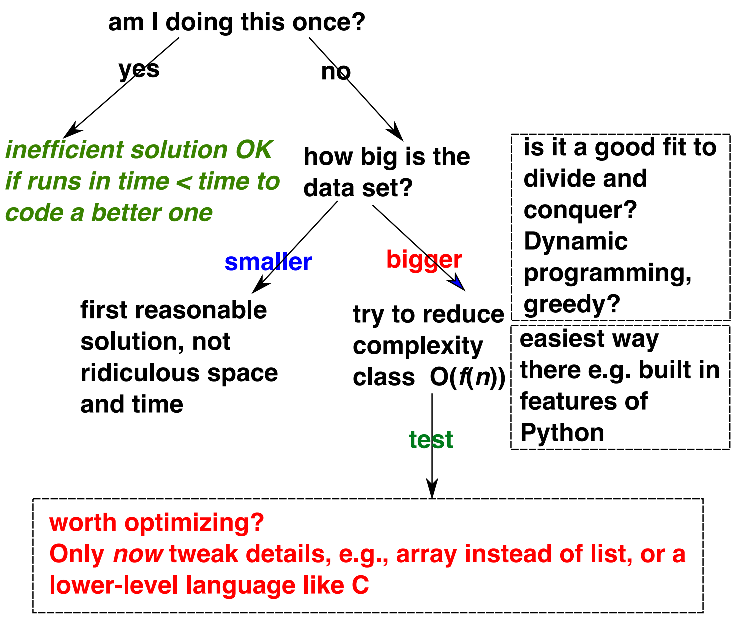
Last typeset 4 June 2019

Preface

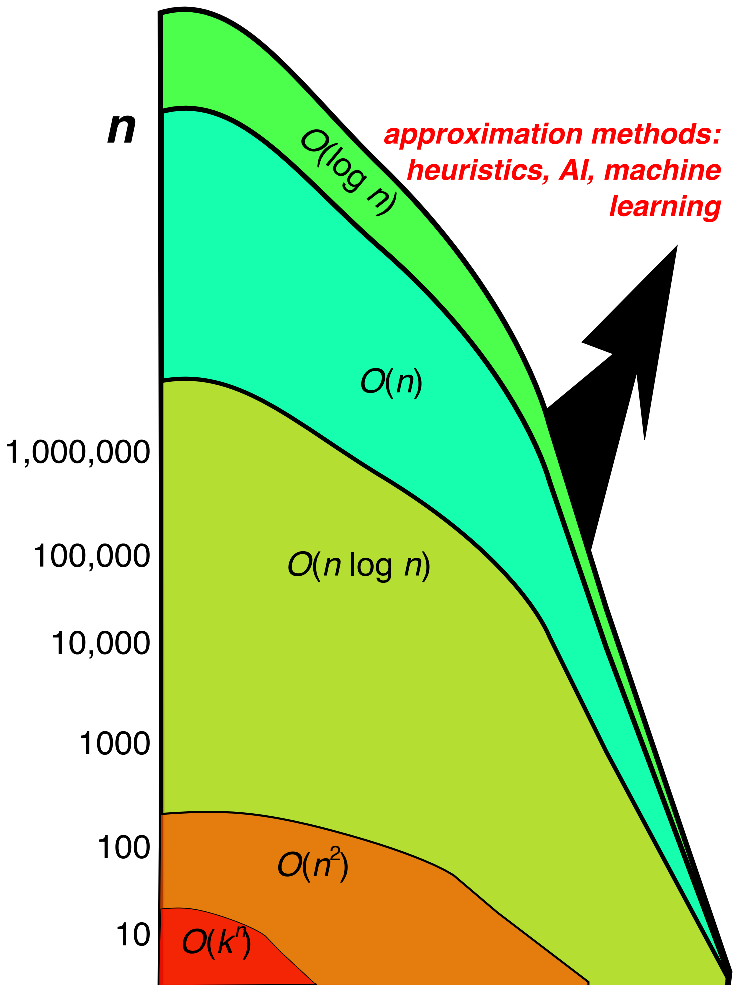
WHY THIS MATERIAL? Bioinformatics is a difficult subject because it integrates so much from multiple disciplines. The emphasis here is on algorithmic thinking – working from a problem to an implementation while thinking *analytically* about efficiency concerns.

The picture illustrates a general plan for algorithmic thinking. Anything that can be classed as an algorithm can be analysed and your design choices are *not* always to find the most efficient algorithm possible.

The aim is to solve a problem as efficiently as possible; if it is something you do only once, that results in a rather different set of choices than if you are going to do it many times. And – of course – size counts. That is what this course is


am I doing this once?
yes
no
inefficient solution OK if runs in time < time to code a better one
how big is the data set?
is it a good fit to divide and conquer? Dynamic programming, greedy?
smaller
bigger
first reasonable solution, not ridiculous space and time
try to reduce complexity class O(f(n))
easiest way there e.g. built in features of Python
test
worth optimizing? Only now tweak details, e.g., array instead of list, or a lower-level language like C

Generic Design Decision Logic: guidance on how to apply algorithmic thinking to real problems.



Efficiency decisions: the area of each zone roughly indicates how many useful algorithms are likely to be found in that zone. Some popular bioinformatics tools like BLAST fall in the "approximation" zone.

about – how to think through algorithmic problems and arrive at useful solutions.

The next picture illustrates roughly how problem size relates to algorithm performance categories called *complexity classes* (see page 8). If the complexity class is a very rapidly-growing function as n, a measure of problem size, grows, an algorithm in that class is not useful except for small values of n. Understanding that concept is a key part of this course. Boundaries are very approximate and depend on details of the algorithm, how long you are prepared to wait and whether it's possible to find an algorithm in a more efficient class.

As I said at the start, size counts...

Contents

List of Figures

List of Tables

Listings

1 Introduction

ALGORITHMS AND DATA STRUCTURES are a fundamental part of computer science and any understanding of bioinformatics that goes deeper than elementary use of standard tools and techniques requires at least some understanding of these concepts. Developing new tools and techniques requires a deeper understanding. One of the most critical concerns in design and analysis of algorithms and data structures is *scalability*. Without the right analytical tools, it is difficult to predict how performance will vary when you move from small test cases to big data sets and bioinformatics is a field with very big data. The human genome, for example, is about 3 Gbytes, represented the most obvious way, with a single letter for each base.

To start, I explain some of the fundamentals of what we study in data structures and algorithms, then go on to how this all applies to bioinformatics, to set the scene for what follows.

The take home message? *A box of this kind reminds you of key points.*

Heads up: *A box like this warns you of things that may not be obvious or that could catch you out.*

1.1 Basics

First, what is an *algorithm*? An algorithm is a precisely-defined method for obtaining a particular result that should eventually conclude with the correct answer. Because an algorithm is *precisely defined*, it is possible to implement it in any computer programming language. Provided it is correctly implemented, the program containing the algorithm will eventually finish (because an algorithm should always *terminate in a finite amount of time*) and it should produce the *correct* answer. In summary, the properties of an algorithm are:

1. *precisely defined* – there should be no ambiguity about what it does
2. *terminate in finite time* – it should eventually finish
3. *correct* – it should produce the correct result for its purpose

Any computer program that implements an algorithm should therefore produce the correct result in a finite amount of time.

In practice, there are constraints; a “finite” amount of time can still be too long to be practical and a real computer has limits like the size of its memory.

Second, what is a *data structure*? A data structure is a method of storing information that can either be *single* item or some kind of *composite* item. Composite data structures can be a group of individual items each identified by *name*, a collection of items each identifiable by either a *key* or their *position* in the data structure. An example of a key is a name used to identify a person in telephone directory, or a student number used to identify a student in a university.

Aside from how data is stored, a data structure is also characterised by *operations* on it – for example:

- how two or more of them can combine (e.g., strings can be *concatenated*)
- whether arithmetic on them makes sense and, if so, how it is defined

Scalability

The study of scalability falls into two sub categories: space complexity and time complexity.

- *space complexity* is concerned with how much more memory an algorithm needs than that used to store the data.
- *time complexity* is concerned with the time it takes to run an algorithm.

In both cases, complexity is a expressed as a function of the size of the data, usually taken as the number of items in a collection. For example, if a genome has 3-billion bases, the size of the data is 3-billion, without considering how each base is represented. What we are interested in is measuring how time (or additional space) grows as we scale the problem size up.

If, for example, an algorithm's time complexity is a function in which the biggest term is n^2 (quadratic), it will scale much more poorly than an algorithm whose time complexity has no term bigger than n (linear).

What concerns us is how fast time (or space) grows as n grows, not the precise formulation of the total space or time required. A function describing time complexity that looks like n^2 means that if you double the size of the problem, you need not *twice* the time to complete but *four times* the time, since $(2n)^2 = 4n^2$. Because we are concerned most about growth rate, we generally drop constants as well as lower-order terms.

To emphasize that we are only looking at the highest term and ignoring constants, we write the complexity as $O(f(n))$ where f is the function of interest, e.g., $O(n^2)$. We can define the notation more formally using limits:

$$f(n) \in O(g(n)) \text{ if } \lim_{n\to\infty} g(n) \leq k \times f(n) \text{ for sufficiently large } k \tag{1.1}$$

This definition says that we can choose $g(n)$ to be an increasingly good approximation to $f(n)$, disregarding constants, as n grows; we can also define variants on the notation where $g(n)$ is a strict upper or lower bound (never equal). The $=$ part of the inequality implies that we should ideally choose g so that $k \times g(n)$ could be approximately equal to $f(n)$ for suitable k and big enough n; the $<$ allows the possibility that it may be inexact. In practice, we choose g so that it is as simple as possible and captures the fastest-growing part of f. Note that $g(n)$ defines a *set* of functions; we can choose the one that best characterizes a complexity class and any actual function that meets the definition is a member of the same set.

Reading $O(f(n))$ as "order $f(n)$", meaning that $f(n)$ characterises how fast time (or space) required grows, suffices for our purposes. This notation is sometimes called "big-O", since $o(f(n))$ is also used in some scenarios, but we can abbreviate it in conversation to "O" (pronounced "Oh")[1].

When we look at space complexity, as with time complexity, the concern is how space required grows as the data size grows – but the space we are concerned with is *over and above* that required for the original data (extra variables etc.) since the space for the original data is needed no matter what the algorithm is.

[1]To be strictly formal, the "o" and "O" are actually the Greek letter omicron in lowercase or capitals; there is another variant on describing the complexity class that uses the Greek letter omega in lowercase or capitals.

A particular concern is with *recursive* algorithms, since each layer of call needs storage, even if you do not explicitly create that as a variable.

> **The take home message?** *Space and time complexity are concerned with growth rates as the problem size scales, not absolute values of time or space needed. We use big-O notation to separate out the term that most strongly affects growth.*

> **Heads up:** *Space complexity is not concerned with the memory required to store the data, but* extra *memory needed by the algorithm. We ignore space complexity unless it is a function of data size, n (e.g., a few extra fixed-size variables don't count).*

Best, worst, average

Another thing we need to consider with algorithm analysis is how much complexity can vary *for a given problem size*. Some algorithms may have a *best case* that is a lot worse than the *worst case*, and an *average case* that could be somewhere between. For example, some sorting algorithms (see §3.2) have a best case of $O(n)$ and a worst case of $O(n^2)$. There is at least one with a best case of $O(n \log n)$ and a worst case of $O(n^2)$. Yet the latter is one of the best sorting algorithms because its $O(n^2)$ scenario can (usually) be prevented, whereas the algorithm with a best case of $O(n)$ only does that well in a special case and $O(n^2)$ is more typical behaviour.

Average-case analysis requires arguing about how the algorithm varies across all possible data variants (hence, all variants on the algorithm's behaviour) and can be difficult unless this sort of variability is easy to characterise. In a case where the best and worst analyses are very different (e.g., $O(n)$ *vs.* $O(n^2)$), the average case is highly informative in deciding whether this is a good algorithm for large n. If the average is also $O(n^2)$, this is not a method you are likely to want to use for large data sets. If, on the other hand, an algorithm with a best case of $O(n \log n)$ and a worst case of $O(n^2)$ has an average case of $O(n \log n)$, it will be a reasonable algorithm to use if the worst-case behaviour is possible to avoid.

It is important to understand that best, average and worst refer to variations for the same n and as usual the big-O notation is of interest to describe how algorithms scale up as n grows – so it is *not* the "best case" when n is small.

The take home message? *Understanding conditions that lead to best and worst case analysis is important for understanding which or both is relevant in a given context. Average-case analysis is often harder but can be useful when there is a big difference between best and worst analyses.*

Heads up: *Best, worst and average refer to variations for given n.*

Before we get to a specific algorithm, we can also analyze properties of the problem to get an idea on whether it is solvable. Two concerns are

- *computability* – whether it is possible to solve the problem at all
- *tractability* – whether it is possible to find an efficient solution to a problem

Computability

Computability theory is a large and complex area. It does however contain some elementary results, the best known of which is the *halting problem*. To give a sense of the subject, I explain that one example. For the most part, we are not addressing problems in bioinformatics where computability is a concern.

The halting problem is a *decision problem*. Stated simply, it is an algorithm $H(G,I)$ that decides if, given algorithm G and a particular input to that algorithm I, $G(I)$ will run to completion (halt). We can prove it is not possible to find a general algorithm H as described, using *proof by contradiction*, a proof technique that shows that an assumption leads to a contradiction (in logic, **true** = **false**).

In this case, we assume we can find algorithm H (returns **true** if given as input an algorithm that terminates on a given input, **false** otherwise) and construct algorithm F that takes algorithm X as input follows:

```
F (X) :
  if H(F, X)
     loop forever
  else
     terminate
```

We have two cases if we run this algorithm:

1. $F(X)$ *terminates* – then calling $H(F,X)$ will give a **true** result and $F(X)$ *doesn't* terminate

2. $F(X)$ *doesn't terminate* – then calling $H(F,X)$ will give a **false** result, resulting in $F(X)$ terminating

So the assumption that $H(F,X)$ exists leads to a contradiction in both cases. This is a particularly elegant proof showing that a whole class of algorithm is not possible; all it requires is the conceptual leap of regarding an algorithm as data that can be examined by another algorithm.

What is really remarkable about this proof is that it originated – in a slightly different form – in 1936, before any computers existed [Turing 1937][2].

Tractability

Another issue in practicality of finding an algorithm for a given problem is the theoretical limit on how efficiently the problem could be solved. What we are looking at here is the *problem*, not the solution.

Sometimes, we can prove that a problem can *never* be solved in better than exponential time ($O(k^n)$), which means it will never be solvable in a reasonable amount of time no matter what algorithm we find. One example is trying to play chess by looking ahead at every possible move. Since a chess game has something like 10^{62} variations in board positions (based on an average of 35 possibilities per turn and an average game length of 40 moves[3]: $35^{40} \approx 10^{62}$), this is never going to be a workable solution. Even if you could compute a trillion moves per second, it would still take about 10^{43} years to compute all the outcomes. You need not even try to find an algorithm. Here, the power is a constant rather than a function of n but is so large that the problem space is too big to search.

Generally speaking, for modest-sized n, polynomial-time problems, particularly when the biggest power is 2, are reasonably tractable. As n grows, even these quickly become too slow.

An in between case is problems where the best-known solution takes exponential time but there is no proof that a polynomial-time algorithm does not exist. An important class of such problems is called NP-complete. We will not explore

[2]The author of this paper, Alan Turing, was at the forefront of developing the first electronic computers in World War II, used for breaking German codes. This project was top secret and only made public in the 1970s, so a US project, ENIAC, completed too late to be of use in the war, was credited with producing the first working computer. More here: http://www.colossus-computer.com/.

[3]Some discussion of that here: https://chess.stackexchange.com/questions/2506/what-is-the-average-length-of-a-game-of-chess.

this concept in any detail: it is just important for our purposes to know that some problems can never have efficient solutions, unless we use an approximation method that is not guaranteed to find the best answer.

The most scalable algorithms are generally better than polynomial time: $O(n \log n)$, linear ($O(n)$) or logarithmic ($O(\log n)$) time. Generally, we characterize an algorithm by a function like this that places it into a *complexity class*.

In most algorithms work, problems that are NP-complete or known to be of exponential time complexity are considered *intractable* and can only realistically be solved for small n or using approximations.

Alternatives to Algorithms

Sometimes an algorithm either can't be found or is inherently inefficient – the problem may be intractable or unsolvable, or the data size simply too large even for a "tractable" algorithm. In bioinformatics, we often encounter the last case where the time complexity would be reasonable for a moderately large data set but e.g. for a whole genome or for multiple-genome problems, $O(n)$ or $O(n \log n)$ is not practical.

In these situations various approximation techniques apply, some derived from the field of *artificial intelligence* (*AI*). One example is the use of *hueristics*, rules of thumb that guide to a solution without guaranteeing that the solution is correct or optimal. Another is *machine learning*, whereby patterns of or classifications of data are found by methods that are inherently approximate.

These approaches lose one key property of an algorithm, a correct solution (or, more accurately, they may not produce a *completely correct* or *optimal* solution). The gain is solving a problem where an algorithm would either not be able to find a solution or would take too long.

The take home message? *Using heuristics, machine learning or other techniques from AI makes it possible to solve problems that would take too long or even not be solvable at the expense of losing accuracy, optimality or the assurance of correctness.*

1.2 Algorithm Design Styles

Algorithms often can fit into one of several common patterns; spotting one of these aids in analysis and provides a starting point for design. Common examples are divide and conquer, greedy and dynamic programming.

Divide and Conquer

The divide-and-conquer approach is based on breaking a problem into smaller pieces until you can solve it directly. This method is a natural fit to *recursion*: if you are not at a case that can be handled directly, invoke the algorithm again on one or more smaller instances of the problem until you break it down to a case that you can directly handle, then pass the solution up to where the algorithm called itself.

A simple example is *binary search*, where you have data in an array that is sorted and you search for a particular value. A recursive algorithm for binary search looks like Listing 1.1.

```
binsearch (data, key) -> boolean
   N = data.size
   if (N = 1)
     if data[0] = key
       return true
     else
       return false
   if data [N/2] <= key
     return binsearch (data[N/2..N-1], key)
   else
     return binsearch (data[0..N/2-1], key)
```

Listing 1.1: Pseudocode: recursive binary search

This is an example of *pseudocode*: a program-like notation that need not be exact on details, as long as it expresses the algorithm clearly enough to convert to code. Here, I assume arrays start on index 0 and I can make a *slice* of an array by giving the range of the start and end index. Here is the equivalent code in Python in Listing 1.2. Note the use of a *docstring*, with triple-quotes (double-quote symbols appearing 3 times) to open and close. This makes documentation about a function possible to extract easily, and is good Python programming practice.

```
def binsearch (data, key) :
    """binary search example to illustrate recursion
       pass in: sorted list to search, key to search for
       return True if found False if not
    """
    N = len(data)
    print key," in ", data," at ", N/2
    if N == 1 :
         if data[0] == key :
           return True
         else:
           return False
    if data [N/2] <= key :
         return binsearch (data[N/2:], key)
    else:
         return binsearch (data[:N/2], key)
```

Listing 1.2: Python: recursive binary search

Python has some syntactic details you have to get right like indentation, colons at the end of particular lines and spelling `True` and `False` with an initial capital. These annoying little details are necessary so your program can run; on the other hand, Python has a very neat syntax for slices (on lists, in this example – they look at bit like arrays but are not, see §2.2.2): the notation `data[N/2:]` (everything from element $\frac{N}{2}$ on) and `data[:N/2]` (everything up to but *not including* element $\frac{N}{2}$) divides the data so it is all included in either the one partition or the other. By contrast, with my pseudocode notation, I had to be very explicit about what was in each slice of the data, since my notation is meant to be obvious just by reading. The Python notation is not obvious but is easier if you know what it means.

For comparison, Listing 1.3 contains a Python version of an iterative (non-recursive, using a loop) version of binary search. It is conceptually simple, but the details are tricky to get right.

```
  def binsearchloop (data, key) :
    """binary search example using a loop
       pass in: sorted list to search, key to search for
       return True if found False if not
    """
    N = len(data)
    low = 0
    high = N-1
    while low <= high:
       mid = (low+high)/2
```

```
        if data [mid] == key :
          return True;
        if data [mid] < key :
           low = mid+1
        else :
           high = mid-1
     return False;
```

Listing 1.3: Python: iterative binary search

So in this case, there is a small win for recursion. Both approaches are divide and conquer – recursion more obviously fits the mould, but the iterative version also cuts the data set down until there is an exact match to a comparison.

The cost of the recursive solution is an internal data structure created for you called the *stack* that keeps track of the depth or recursion and allows you to return to where each call originated. For this particular case, the stack can grow to at most $\log_2 n$ deep for search n items, which is acceptable even for very large n. That is an example of *space complexity*. Here, the space complexity is $O(\log n)$. We need not give the base of the log as logs in two bases differ by a constant factor, according to the base conversion formula:

$$\log_a X = \frac{\log_b X}{log_b a} \tag{1.2}$$

What of time complexity? The algorithm in both forms can be proved to take at most $log_2 n + 1$ steps, which is also $O(\log n)$.

The take home message? *Recursion can be a powerful technique and is a natural fit to divide-and-conquer problems.*

Dynamic programming

Dynamic programming is a *bottom-up* method in which you solve small subproblems until you solve the original problem. Since you start directly with subproblems rather than the *top-down* approach of divide and conquer, dynamic programming can be more efficient in a case where there are many repeated subproblems. By keeping a table you build of previous solutions you can build on the solutions already known. In that scenario, divide and conquer can be very wasteful as it recomputes every subproblem each time it is encountered even if has been solved before.

The value of dynamic programming is really only obvious with relatively complex algorithms; I give an example here of a relatively simple one: *approximate string matching*. My algorithm is based on Baase and van Gelder [2000, p 507] but corrects an error in their description.

Figure 1.1 illustrates the core idea of the algorithm. You have two strings you are comparing, a shorter search string s length m and a longer text t of length n. You want to find a *k-approximate match*, the first place in the longer text where the search string is an exact match except for up to k differences. You can distinguish four cases:

- *match* – the two characters currently being compared match
- *revise* – the two characters currently being compared differ
- *insert* – the search string has an additional character
- *delete* – the search string has a character missing

If you are perceptive, you will note that these alternatives are not mutually exclusive; did a mismatch occur because a character was inserted, deleted or altered? The solution to this dilemma is to score all the alternatives if there is a mismatch and build a measure of mismatch that minimizes over all the alternative ways of scoring a mismatch.

At each position, we score the four alternatives and the *difference matrix* (called D) entry for position i in the search string and position j in the text is the minimum score for each of the above alternatives. What figure 1.1 illustrates is how we can compute $D[i][j]$ purely using nearest neighbours: each position where either or both of i and j is reduced by 1.

The score at position $D[i][j]$ is calculated as the minimum value of calculations 1, 2 and 3:

1. choose one of these to compare with the other two values:
   - $matchCost = D[i-1][j-1]$ if $s_i = t_j$
   - $reviseCost = D[i-1][j-1]+1$ if $s_i \neq t_j$
2. $insertCost = D[i-1][j]+1$
3. $deleteCost = D[i][j-1]+1$

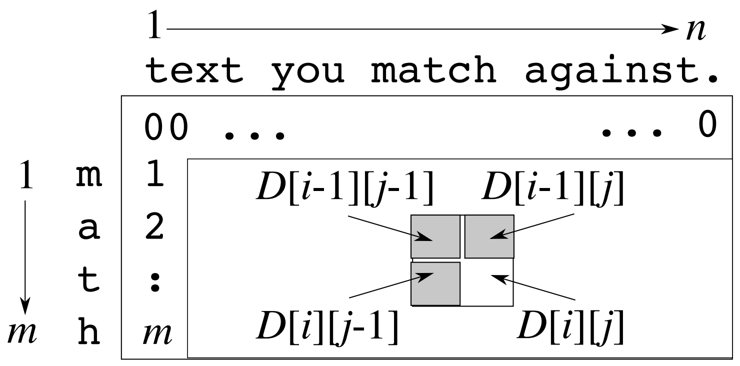


Figure 1.1: Partial string match matrix. The search string (length m) is down the side; the text to compare (length n) along the top. The top row is initialized with zeroes and the first column with $0 \ldots m$.

You find a k-approximate match if a value $\leq k$ appears in the bottom row and you can stop at that point.

How does it work?

As you go down column i, the number at position $D[i][j]$ represents how many mismatches there are in the substring of t up i, when you have looked at the shorter string s up to position j. Look down column 0: at the top, the entry is 0 meaning you have scanned 0 positions of both strings, so there are 0 mismatches. Go down 1 row and the value is 1: this means you have looked at 1 position of the search string s and none of the text t, so there is 1 mismatch. Go down 1 more row, and the value is 2 because you have looked at 2 positions of the search string s and none of the text t, so there are 2 mismatches. That is how you develop the first column – it is initialized like this rather than created as the algorithm runs. The top row is also set to all zeros.

Start the second column now; this is where the algorithm really starts. If you compare the first element of s with the first element of t (or $i = j = 1$), you can now apply the algorithm. If the two are equal, use $matchCost = 0$ ($D[i-1][j-1] = D[0][0] = 0$); you will find the minimum of this and *insertCost* and *deleteCost*. Since the latter 2 are both 1, the minimum is 0, so $D[1][1] = 0$. If on the other hand $s_1 \neq t_1$, you will use *reviseCost*, which is also 1, so $D[1][1] = 1$.

If you keep going like this the number in the bottom row is how many differences there are between s and the substring of t ending at that column of the table. So to find the first k-approximate match, keep going until you find a column with bottom entry $\leq k$.

Although the algorithm conceptually needs a table with $(m+1) \times (n+1)$

entries, in fact it is only necessary to keep two columns at a time as generating the current entry only needs those above it (calculated in the same pass) and those to its immediate left (calculated in the last pass). Its space complexity is therefore $2(m+1)$ or $O(m)$, acceptable since usually $m \ll n$.

Time complexity is $O(mn)$, which is good particularly if the smaller string is short (length m). However for things like sequence alignment, the algorithm is not efficient enough as a very large number of runs of the algorithm could be needed and sequence alignments frequently compare lengthy sequences with both pairs roughly equally long. Aligning multiple sequences scales the problem up further.

Greedy algorithms

A greedy algorithm, like dynamic programming and divide and conquer, works by solving subproblems. However the approach differs in attempting to find the *most promising subproblem* to tackle next rather than simplistically starting at one end of the problem and working through from there.

There are cases where the greedy approach only produces an approximate or suboptimal solution (such a solution is not strictly an algorithm) but there are cases where greedy algorithms work very well (getting the correct solution fast).

Since the greedy approach grabs a chunk of the problem to solve, leaving a smaller subproblem to solve, it is more likely to result in a top-down than a bottom-up approach.

The ideal case for a greedy approach is where it results in a correct, optimal algorithm. Where this works, it can be a very efficient approach. However, grabbing the most promising subproblem is not always the best approach. For example, if you are trying to make change with notes or coins of a limited range of values, a greedy approach may leave you unable to solve the problem even though you could with a different choice of smaller denominations. To illustrate this, assume you have a till full of $2 notes, one $5 and nothing else. You need to make change of $6. Start with the greedy choice, $5. Now you are stuck. If instead you considered all options, you could have made change with three $2 notes[4].

[4]For US readers, yes, there *is* a $2 bill: `https://en.wikipedia.org/wiki/United_States_two-dollar_bill`. The fact that many people do not believe it exists would be a reason for this denomination to accumulate in a till.

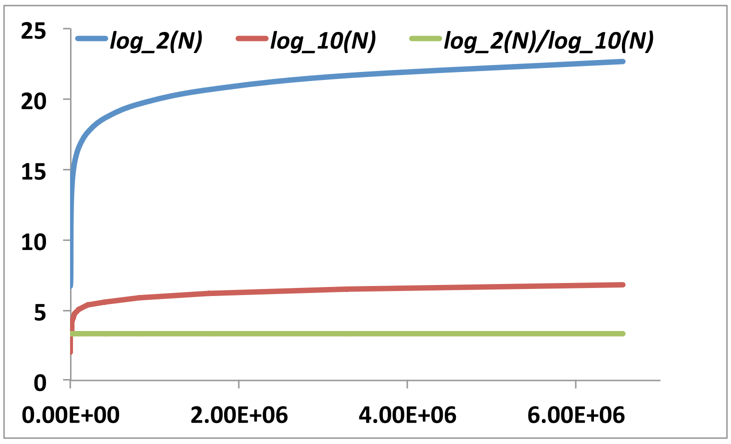


Figure 1.2: Comparison of logs to different bases: they differ by a constant factor.

1.3 More Complexity

When we look at time and space complexity, because we are only concerned with how fast they grow as n grows (sometimes a function of more variables, but calling this n is sufficient for general principles), we can often omit inessential details. For this reason, a pseudocode notation that is imprecise is acceptable, as long as details we omit do not obscure any contribution that *is* a function of n and could change the *asymptotic growth rate*.

Though I did define the asymptotic growth rate more formally using limits, what we are concerned with is the biggest term in the function describing complexity (space or time). Since we are concerned with growth rate, we can drop any constant multiples.

To take an example, I mentioned that we are not concerned with the base of a log because all logs of different bases are related by a constant factor. Figure 1.2 illustrates this point. The horizontal line is a constant: it is the ratio between the two log functions.

Now check out Figure 1.3. The point I make here is that a very simple algorithm with poor time complexity may do better for small n than a far more complex algorithm with much better time complexity; lower time complexity wins out in *bigger* examples. The functions describing the behaviour of the algorithms are $100n^2$, $1000 \times n \log_2 n$ and $1000 \times n \log_2 n + 10000 \times n$. The last function takes a while longer than $1000 \times n \log_2 n$ to start beating the $100n^2$ function, but it does in the end. What this example illustrates is that except for relatively

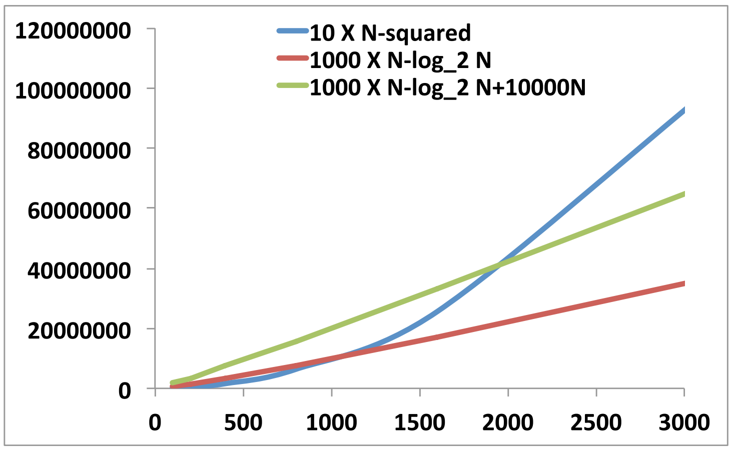


Figure 1.3: Comparison of functions with different complexities: higher complexity loses despite lower constants for n large enough.

small examples, it is worth concentrating on finding an algorithm with the lowest possible time complexity, in this case, it appears, $O(n \log n)$. Once you have that, you can focus on making it more efficient – in this case, maybe the two variants on $O(n \log n)$ algorithms represent different levels of attention to detail. There is a lot less to gain tweaking one of them than in moving away from $O(n^2)$. Even so there is value in optimizing the slower $O(n \log n)$ algorithm; working hard on making the $O(n^2)$ algorithm faster will not be worth significant effort.

Sometimes, it's worth exploiting knowledge that a simpler algorithm is faster for small n. For example, there are sorting methods that reduce the problem to smaller subproblems (divide and conquer) then use a simpler sort method with poor complexity once the problem size is small enough for that method to be reasonable to use. When would that be? We can measure the cross-over point and switch to the simpler algorithm when it is clearly faster. For example, some sorting methods are $O(n^2)$ but so simple that they are much faster than any $O(n \log n)$ for $n < 100$. So some $O(n \log n)$ methods reduce the data size to the point where the $O(n^2)$ algorithm is faster, use the simple algorithm on that subset of the data then combine the sorted sections of the data using the more scalable algorithm. This trick works well with divide and conquer, since part of that strategy is combining solutions to smaller sub-problems.

In summary: it is clear that finer detail of an algorithm is not always necessary to do time and space complexity analysis, so I use pseudocode when it I do not need to illustrate implementation details.

The take home message? *Lower complexity wins for larger n but a simple algorithm can still be good when you know n is small e.g. to solve a problem of known size.*

1.4 Further Reading

Cormen et al. [2009] is a very good general algorithms text with more detail than you are likely to need. There are many other algorithms and data structures books, either general [Baase and van Gelder 2000] or specific to computational biology [Elloumi and Zomaya 2011]. Also search for algorithms on the Internet; Wikipedia is a good starting point but Wikipedia articles should always be checked against authoritative sources.

Exercises

1. Using the definition 1.1 of big-O on page 4:
   (a) is $n^2 \in O(\log n)$?
   (b) is $\log n \in O(n^2)$?
   (c) is $3n^2 + 2 \in O(n^2)$?
   (d) which of these is useful if true? Why?
2. We store data in an array and want to sort it using an algorithm that uses exactly 12 extra bytes of variables no matter how many items we sort. If the array holds n items, what is the space complexity of the algorithm? Explain.
3. Someone asks you to write a piece of software that will check any program to see if it runs forever or correctly terminates. Is this something possible to do? Explain.
4. Explain why it is not practical for a computer to play chess by evaluating every possible board position.
5. A particular greedy approach sometimes gives good answers quickly but cannot guarantee to do so: sometimes the answer is inaccurate. Can this strictly be called an algorithm. Explain.

6. Compare and contrast greedy, divide and conquer and dynamic programming algorithm design strategies.

7. Can you think of a biological application for *k*-approximate string matching?

2 Data Structures

A LANGUAGE LIKE PYTHON provides a wealth of built-in data structures so if you use that class of language, you seldom have to build data structures from scratch. However, an understanding of what is happening underneath you is useful so that you do not make inefficient choices. In this chapter I describe common data structures, show how they can be built up and illustrate how Python provides them, usually as a high level construct.

Related to data structures is methods of access, which vary according to the data structure. In general, a data structure can be characterised by:

- *structure* – how it is organized
- *access* – how elements can be accessed from a starting point
- *search* – how elements can be accessed without a starting point

Some aspects of data structures emerge from how we use them so this chapter is a starting point for successive chapters.

2.1 Basics

The simplest data structures are those that permit access in a particular sequence. Next are variants on those that can be accessed by a specific value – either an *index* (a numeric position) or a *key* (any kind of value). Finally, there are data structures that have a more complex interconnection between elements – in general, graphs (of which trees are an example).

Some of these data structures are built into programming languages, with a fair amount of variation between languages.

2.1.1 Arrays

For example, an *array* is a data structure that has elements the same size and that can be indexed using an integer indicating how far along you are from the start of the array. A common convention is for array indexing to start on zero[1]. An array is based on machine-level addressing where it is relatively easy to find an item *i* places from the start if all items are the same size. Some languages apparently have arrays where elements are not the same size; they do this by storing the actual contents somewhere else and using a *pointer* to the actual data within the array. A pointer (sometimes called a reference in languages a bit further from the machine) is essentially a *memory address*: a number indicating where in memory something is stored.

2.1.2 Lists

An array is close to machine level; a list provides more abstract operations and is a basis for creating more complex structures like trees.

A list has several basic operations common to all languages with that have a predefined list concept:

- head – get the first item
- tail – get a list containing all but the first item
- traverse – process each element

These basic operations have different names and notations in different languages. The key thing to understand is that lists are very efficient at *sequential* access.

2.1.3 Graphs

A graph is a data structure in which *nodes* (also called *vertices*; node is a more obvious term than *vertex* but the two terms mean the same thing) are connected by *edges*. Few languages have a graph as a built-in structure; it must be built out of lower-level components.

Graphs have differing properties:

[1]A few languages differ; most you encounter will use 0-based indexing. Some languages leave the index range up to the programmer; others start it on 1 instead of 0.

- *connected* – a graph in which every edge is connected to every other edge, possibly indirectly
- *directed* – a graph in which edges have a direction is a *directed graph* or *digraph*; the direction can be one way or bidirectional
- *cycles* – a graph with a cycle has a node that can be reached through a path from itself; this is only a concept in digraphs
- *degree* – number of edges connecting a node; in a digraph
  - the *in-degree* is the number of connections into a node
  - the *out-degree* is the number of connections out of a node

Trees

A *tree* is a special case of a digraph *without cycles* – it has an in-degree of exactly one and an out-degree of one or more. A special case is a *binary tree*, with an out-degree of at most two.

A tree has its own special terminology:

- *descendant* – a node reached from another node
- *parent* – the node above another node (i.e., the one from which it is an immediate descendant)
- other relationships are sometimes named as well:
  - *child* – the node descended from another
  - *sibling* – a node with the same parent as another
  - *uncle* or *aunt* – a node that is a sibling of the parent
  - others e.g., *cousin*, *grandparent* can be used if the meaning is clear
- *root* – the only node with no parent
- *leaf node* – a node with no descendants
- *interior node* – a node with a parent and at least one descendant (not the root or a leaf node)
- *height* – the longest path from the root to a leaf

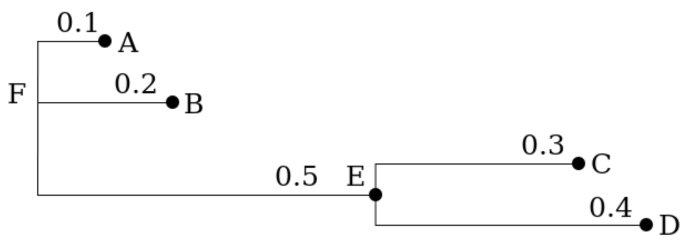


Figure 2.1: Phylogenetic tree example.

Ironically a tree is usually depicted with the root at the top and the leaves at the bottom.

How can a list represent a tree? Consider the following representation of the phylogenetic tree illustrated in Figure 2.1 called Newick format)[2]:

```
(A:0.1,B:0.2,(C:0.3,D:0.4)E:0.5)F;
```

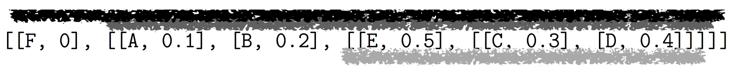
You could represent this as a list as follows (using Python notation – loosely, see §2.2 for Python notation done more correctly):

```
[[F, 0], [[A, 0.1], [B, 0.2], [[E, 0.5], [[C, 0.3], [D, 0.4]]]]]
```

Here I rely on the fact that a list element can be another list. Each element of the list is either the root of the tree (the first element) or another list of descendants. If a contained list is a leaf node, it only contains its name and the distance from its parent. Otherwise, a level that is not a leaf level is represented as a list of the current level's nodes and its descendants. For example, the descendants of F are A, B and E so they are all in the list defining the node following F. C and D are the next level down so they are inside another layer of list, the list containing E.

In summary, a tree is represented as a list. All nodes are represented as a list, a pair containing name and distance from the root. A leaf node is only a list containing the pair representing that node. An interior node is contained in a list containing the current node followed by its descendants. Figure 2.2 illustrates how each level of the tree is represented as this list – the top level containing the

[2]Source: https://en.wikipedia.org/wiki/Newick_format.



```
[[F, 0], [[A, 0.1], [B, 0.2], [[E, 0.5], [[C, 0.3], [D, 0.4]]]]]
```

Figure 2.2: A tree represented as a list.

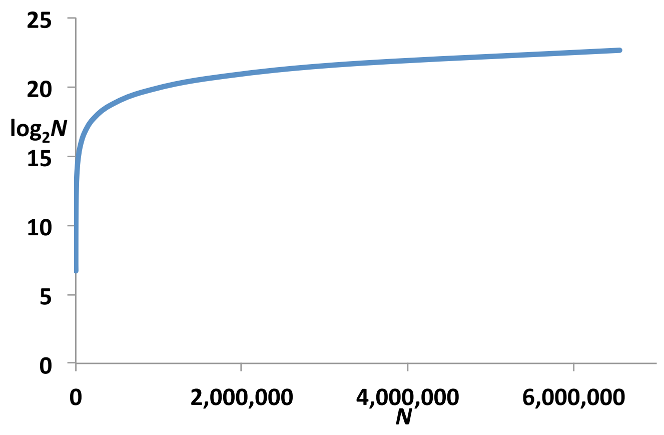


Figure 2.3: Growth of $\log_2 N$.

root is highlighted with dark shading; the subtree descended from that in lighter shading and the subtree a further level down with the lightest shading.

Aside from phylogenetic trees, which have wide use in biology, a tree is a useful data structure for finding information efficiently. A *binary tree* has at most 2 descendants (out-degree 2) and an *ordered binary tree* is a binary tree in which data is inserted and hence can be found using a specific ordering. The usual way of doing this is:

- $<$ *root* – insert to the left
- $\geq$ *root* – insert to the right

In the ideal case, where the tree is *balanced*, searching for a value out of N values in a tree takes at most $\log_2 N$ steps. To define this concept we need another definition. A tree is:

- *perfect* if all non-leaf nodes have exactly 2 descendants and all leaves are at the same level
- *balanced* if it is of height k and it would be perfect if all leaves at level k are removed

A balanced tree has other definitions that may differ slightly; the key point is that the difference between the shortest and longest path (height) is at most 1.

A perfect tree of height k has 2^k leaves and $2^{k+1} - 1$ nodes. We can prove this by induction.

Table 2.1: Adjacency matrix example. Each row represents nodes that the named node connects to, based on the graph of Figure 2.4. A "0" means no connection in the direction from the node named in the row to the node named in the column heading, a "1" means a connection.

	A	**B**	**C**	**D**
A	0	1	0	1
B	1	0	0	0
C	0	1	0	0
D	0	1	0	0

- For k=0, a tree has exactly 1 node ($2^k = 2^0 = 1$) and that is a leaf node; $2^{k+1} - 1 = 2^1 - 1 = 2 - 1 = 1$ so the basis case holds.

- For the inductive step, assume the result holds for some $k \geq 0$. Then for $k+1$, we need to show the next bigger perfect tree has $2^{k+2} - 1$ nodes and 2^{k+1} leaves. If height is k and we add 1 to height k, then the extra layer has twice as many leaves as a tree of height k. Since the number of leaves was 2^k this means for $k+1$, the number of leaves is $2 \times 2^k = 2^{k+1}$. The total number of nodes for height $k+1$ is then $2^{k+1} - 1 + 2^{k+1}$. Since $2^{k+1} + 2^{k+1} = 2 \times 2^{k+1} = 2^{k+2}$, the result is proved.

Why is this result useful? Because a balanced tree to a good approximation has a height of $log_2 N$ for N nodes. That means if you can keep an ordered tree of N items reasonably close to balanced, searching for a value in a tree takes about $\log_2 N$ steps. To get some idea how good this is, check out Figure 2.3. It should be clear from the graph that $\log_2 N$ grows really slowly even for large N.

General graphs

More general graphs have various methods of representation, the most common of which is an *adjacency matrix*. The idea is that each node is represented in both a row and a column and a connection is marked where connected nodes' rows and columns intersect. For an undirected graph, only half the matrix is useful as it is symmetrical; for a digraph, a connection may exist in one direction not the other.

Figure 2.4 illustrates a digraph; Table 2.1 illustrates an adjacency matrix for the same graph.

An adjacency matrix is inefficient for a *sparse* graph, one with very few connections compared with the node count. In some cases, an *adjacency list* –

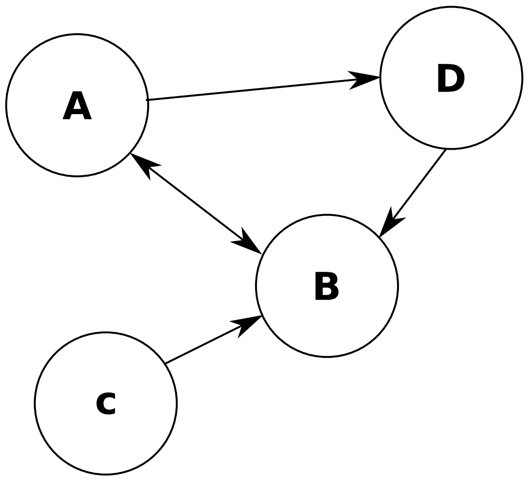


Figure 2.4: General graph. This is a digraph with a mix of edges, mostly unidirectional, one bidirectional.

a list for each node indicating its connections – is more compact. For example, for this graph:

- **A** – `[B, D]`
- **B** – `[A]`
- **C** – `[B]`
- **D** – `[B]`

2.1.4 Content-Addressable structures

An array is accessed by knowing the index of an element. A tree can be used to look up specific values.

If we abstract away the method of storage, a data structure in which we find content by searching for part of that content can be implemented various ways. In general, we divide the content stored into a *key*, the means of identifying the content of interest, and a *value* associated with the key. A $\langle key, value\rangle$ pair implies that the key is only used for identification. However the key could also be part of the value of interest.

A *dictionary* is an example of a data structure where a key (the word) is used to look something up that does not include the key (the definition).

2.2 Relationship to Python

Python has a built-in data structure called a *list*. Python lists have array-like operations such as indexing and *slicing* – as illustrated in Listing 1.2, where I divide a list into smaller pieces. You will be familiar with writing lists between square brackets if you code a lot in Python.

Let us take a look at how Python data structures map onto arrays, lists and trees. Other more sophisticated data structures are directly represented in Python; I look at those afterwards. Lists are a built-in data structure familiar to most Python coders so I focus here mainly on how arrays differ and on how you can build trees out of arrays after a brief look at lists.

2.2.1 Python lists

In Python, the basic list operations can be implemented in various ways. The simplest way to get the `head` and `tail` of a list is to use indexing for the head and a slice for the tail:

```
>>> data = [0, 7, 12, 15, 34, 42, 99]
>>> data[0]
0
>>> data[1:]
[7, 12, 15, 34, 42, 99]
```

There are various ways of traversing a list in Python; the simplest is something like this:

```
for x in data :
   print x
```

The second line can be anything that needs the value “x” retrieved from the list.

2.2.2 Python arrays

Using as list as if it’s an array in Python is convenient and works well for small examples. However, if you want to work with large examples where efficiency is a concern, you need proper arrays where all elements are actually the same size and indexing is efficient. Since arrays are an optional feature, you need to import a module, which you can do like this, to import the array data structure:

```
from array import array
```

Once you have done that, you can create an array like this:

```
x = array('l', [3, 6, 9, 12])
```

where the letter "`l`" represents the type of element, here "signed long", a C type that represents an integer value in 32 bits[3].

2.2.3 Trees in Python

Going back to the phylogenetic tree example, we would have to quote the names of the "species" in the tree to get genuine Python list code:

```
phyloTree = [['F', 0], [['A', 0.1], ['B', 0.2], [['E', 0.5],
   [['C', 0.3], ['D', 0.4]]]]]
```

We can now extract the root of the tree as `phyloTree[0]` – noting that a distance of 0 makes sense if all distances are from the root:

```
['F', 0]
```

and the rest of the tree as `phyloTree[1]`:

```
[['A', 0.1], ['B', 0.2], [['E', 0.5], [['C', 0.3], ['D', 0.4]]]]
```

Since `A` and `B` are at the same level, we can extract them by another level of indexing:

```
>>> phyloTree[1][0]
['A', 0.1]
>>> phyloTree[1][1]
['B', 0.2]
```

What happens if we use `phyloTree[1][2]`?

```
[['E', 0.5], [['C', 0.3], ['D', 0.4]]]
```

This indicates that `E` is an interior node since it is inside a list; the other elements are its descendants.

Here is a Python session that extracts more of the tree:

[3]For a more complete list of array types: https://docs.python.org/2/library/array.html.

```
>>> phyloTree[1][2][0]
['E', 0.5]
>>> phyloTree[1][2][1]
[['C', 0.3], ['D', 0.4]]
>>> phyloTree[1][2][1][0]
['C', 0.3]
>>> phyloTree[1][2][1][1]
['D', 0.4]
```

A few observations; at each level there is a list:

- *root* – only needs to contain its name but for consistency, I add distance 0
- *leaf node* – contains two non-list elements: name and number (distance)
- *interior node* – contains a list with two list elements: a leaf node-like list and a list of descendants

What all this illustrates is that a list is a very general structure, which is why it is built into languages like Python.

2.2.4 Python content-addressable data structures

Python has a *dictionary* data structure built in. A value for a dictionary is written in curly brackets (braces); each $\langle key, value\rangle$ pair is written with a colon between key and value, and a comma separates each pair. For example, if we create this:

```
words = {
  'tree' : 'branching data structure',
  'species' : 'group of living organisms able to interbreed'
  }
```

We can do this:

```
>>> words['tree']
'branching data structure'
```

Once you have started building a dictionary you can add to it as you would assign to an array index, but using a key value instead of an integer, e.g.:

```
>>> words['root'] = 'start node of tree'
>>> words['root']
'start node of tree'
```

A dictionary in Python hides a lot of detail behind the scenes: how can a key be looked up efficiently? How are keys and values stored? Python allows the type of key and value pairs to differ. For example:

```
>>> dictionary = {}
>>> dictionary[42] = 'forty-two'
>>> dictionary['forty-two'] = 42
>>> print dictionary
{42: 'forty-two', 'forty-two': 42}
```

The first line creates an empty dictionary. We then add in the value `'forty-two'` with key `42`, then the value `42` with key `'forty-two'`. In the first case, the value is a string and the key a number; in the second, we reverse that. The Python developers have designed dictionaries so they can encompass that sort of generality and versatility. That generality and versatility comes at a price, as I explore in §2.3.

An advantage of a language like Python is that it hides the detail of all these things; a disadvantage is that it hides the detail of all these things. You can write interesting problems with relatively minimal effort, but it is useful to have a sense of what is going on underneath so you do not use features that are inefficient for your particular problem.

2.3 Case Study: Dictionary

Python's dictionary is provided in the language so you can't alter it (unless you are prepared to make your own implementation of Python). It is however a useful example to use to study how to design a data structure.

In Python, a dictionary is implemented as a *hash table*, a structure designed for fast look up. I start out by explaining a hash table, then illustrate alternatives that have different design trade-offs. Understanding this is useful if you use Python on large data or if you have to do your own coding from scratch in a lower-level language that does not have features like dictionaries built in.

2.3.1 Hash tables

A hash table is implemented in two parts:

- *hash function* – converts the key to a number suitable to use as an array index

- *array* – contains the keys and values

You need to store the key as well as the value since the hash function cannot be guaranteed to produce a different value for every possible key – particularly if you do not know in advance what the key values are. Because of this, a hash function may result in *collisions*, where different keys result in the same hash value and hence map to the same place in the array. There are various methods of handling collisions including storing the collisions in a second table or using a secondary hash function to find another space in the same array.

In the best case, a hash lookup takes 1 step or a number of steps bounded by a constant, and hence time is $O(1)$. However, in the worst case (mostly avoided by careful design of the hash function), a substantial fraction of the data hashes to the same location and handling collisions can result in a linear search, which is $O(n)$. A strategy for making it easier to avoid collisions is to make the hash table significantly larger than it needs to be.

The take home message? *A hash table is the most efficient way of looking up data based on a key provided the hashing function distributes key values reasonably evenly and minimizes collisions. You can rely on efficient implementation of built-in structures such as Python dictionaries.*

2.3.2 Balanced trees

A hash table has the benefit that, provided collisions are minimal, lookup takes constant time (called $O(1)$ in algorithm analysis). However a hash table is inefficient if you want to retrieve the data sorted in key order as that requires sorting. An ordered binary tree allows efficient retrieval of the contents in sorted order but at the cost of making lookup take time $O(\log n)$.

However, a tree is only efficient if it is reasonably close to balanced. In the worst case (e.g., if the data that is inserted is already sorted before insertion into the tree), a tree can be completely unbalanced with all the data down one side resulting in $O(n)$ lookup time complexity. There are various algorithms for keeping a tree balanced that make inserting and deleting much harder to code, though the benefit of maintaining $O(\log n)$ lookup time complexity is worth it.

Two strategies are red-black trees and AVL trees. Both keep a tree close to balanced as new data is inserted or data is removed (deleted) from the tree. Both are too complex for a relatively introductory treatment of data structures and algorithms. You need to be aware that they exist in case you need them; for a language like Python, you can rely on efficient implementation of structures like dictionaries rather than having to build your own from scratch.

A tree wastes less memory than a hash table, provided the data items stored are reasonably big. The main overheads are pointers to the left and right descendants. A red-black tree also needs an extra variable that keeps track of whether the node is considered to be "red" or "black". An overhead of 3 small data items is reasonable if the key or value (or both) is reasonably large. Also the tree is only allocated on demand, whereas the size of the hash table needs to be determined in advance.

The take home message? *For code that is used a lot and meant to be very general the complications of implementing balanced trees are worth the effort. For most of our own coding, simpler strategies are good enough.*

2.4 Further Reading

Hash tables have a long and venerable history [Maurer and Lewis 1975], including applications in bioinformatics [Rizk et al. 2013]. In recent times, hash functions have also been used as a check on whether data has been corrupted [Coron et al. 2005] (you can store the hash value and check if you get it again on what should be the same data – a common application is checks that email and downloaded files have not been tampered with).

Red-black trees go back a long way [Guibas and Sedgewick 1978]. They were preceded by AVL trees [Adelson-Velsky and Landis 1962], named after their inventors, two Soviet scientists GM Adelson-Velsky and EM Landis.

There is plenty of more recent material on these subjects on the Internet, as well as in the books cited in Chapter 1.

Exercises

1. The approach given for organizing a tree into a list is not the only option. Rethink the approach and explain how else a tree could be represented in a list.

2. The STRING protein-potein interaction network database[4] at last access had 2031 organisms, 9.6-million proteins and 1.4-billion interactions.

[4]`https://string-db.org/`, accessed 9 June 2017.

   (a) Is this a sparse graph?
   (b) Is it a digraph?
   (c) Describe differences in how your would implement connections in an adjacency matrix as opposed to an adjacency list.
   (d) Which approach would you choose and why?

3. Is it reasonable to use a Python list in most cases rather than an array? Think of an example where an array would be a better choice.

4. Look up how insertion and deletion work in a normal ordered binary tree and compare with a red-black tree.

5. Would you use a Python dictionary for data you frequently need to retrieve in sorted order?

6. If you want the speed of access of a hash table but also need to retrieve data in sorted order, what could you do if memory use is not a consideration?

7. If you had to write Python code involving phylogenetic trees and you had data in Newick format, would you use that directly, or convert to a format like that of S2.2.3? Explain.

8. Explain why hashing, though the best case of lookup is $O(1)$, has a worst case analysis for lookup of $O(n)$.

3 Sorting and Searching

SORTING AND SEARCHING are common techniques that apply in many areas and I introduce some of the more popular methods. Many programming languages have built-in sorting algorithms; it is not common that you need to build your own so the purpose of this chapter is to use them as examples to illustrate principles rather than to build extensive skills in designing such algorithms.

Sorting nonetheless is a good example of design trade-offs between simplicity for a small example and the value of more complex, scalable algorithms. It also provides an interesting example of a theoretical model for the best possible efficiency of a class of algorithm. Most sorting algorithms assume data can all fit in memory – these are *internal sorting* algorithms. For really large data sets, there are also *external sorting* algorithms that use file space. External sorting is a big subject: mostly you can use existing tools so I only touch on it here.

Searching in its simplest form is a straightforward problem – find a particular value in a data set – that illustrates the value of choosing the right data structures.

3.1 Trees

Trees provide a data structure that facilitates both sorting and searching. Recall how an ordered binary tree places everything less than the root to the left – and does that recursively as you go down the tree. Recall also that if the tree is reasonably close to balanced, that results in a search time $O(\log n)$.

Once the data is in a tree, it can be retrieved in sorted order by visiting each node is defined in listing 3.1 – this order of tree traversal is generally called *inorder traversal*. The name not only captures the fact that you retrieve the nodes in the correct sorted order (for an ordered tree) but that the data processing step is in the middle of two recursive calls.

NULL represents an endpoint in the tree. In a lower-level language like C or C++, this would be a NULL pointer. I use "print" to represent visiting the node in the correct order – some other action could be done instead, depending on the purpose of "sorting" the data.

```
inorder (tree)
  if (tree.left != NULL)
     inorder (tree.left)
  print (tree.data)
  if (tree.right != NULL)
     inorder (tree.right)
```

Listing 3.1: Pseudocode: recursive inorder tree traversal

How do you make sense of this? Logically, everything to the left of the current node should be visited first to visit all the nodes in the correct order since everything less than a given node is inserted to its left. Similarly everything to the right is at least as big as the current node and so, in the correct ordering, can be visited after the current node.

What is the time complexity of this form of "sort"? It happens in two phases: building the tree and traversing the tree.

For building the tree, assuming it stays close to balanced, the longest operation to place a new element in a tree that already contains n nodes is at worst $\log_2 n$ steps. The actual number of steps is $\sum_{i=0}^{n} \log_2 i$, which can be shown to be $O(n \log n)$.

For traversing the tree, each node is visited once on the way down the recursion to the left, again to print and again on the way up from recursion to the right so traversing is $O(n)$.

Overall, then, "sorting" like this is $O(n \log n)$ since adding something like $k_1 n$ onto a function of the form $k_0 n \log n$ will not take it to higher complexity class.

Sorting like this is really only useful if you need your data in a tree anyway because adding in a tree as an extra data structure is extra memory (space complexity $O(n)$), a waste if you don't need it, and the overall time is longer than other $O(n \log n)$ sorts.

The take home message? *Keeping a tree ordered has the benefit that lookups are quick and you can produce the data in sorted order if needed, but understand the trade-offs that make this a suboptimal approach in some cases.*

3.2 More General Sorting

It is worth studying simple sorting methods because they are good enough for small data – and also because some of the more efficient methods are slower for small n and switching to a simpler method for smaller subproblems can be a good choice, as outlined on page 16. So in this section, I start with simpler methods that (aside from special cases) take time $O(n^2)$, then go on to more sophisticated algorithms.

3.2.1 Simple methods

There are many simple sorting algorithms that generally have the property that one value is guaranteed to be in the correct place each pass. That means: if you are sorting n items, you reduce the problem each time by 1. So the problem size on the first pass is n, on the second pass, $n-1$, and so on, down to 1. The amount of work then is some constant times $\sum_{i=1}^{n-1} i$; it can easily be proved that

$$\begin{aligned}\sum_{i=1}^{n-1} i &= \frac{n \times (n-1)}{2} \\ &= \frac{n^2-n}{2} \qquad (3.1)\end{aligned}$$

which is $O(n^2)$. Of the $O(n^2)$ algorithms, *insertion sort* is one of the best because it is very simple and does not move data unnecessarily.

```
insertionsort (data)
   for i in 1 .. N
      j = i - 1
      compare = data[i]
      while j >= 0 and data[j] > compare
          data [j+1] = data [j]
          j = j - 1
      data[j+1] = compare
```

Listing 3.2: Pseudocode: Insertion sort

The algorithm can be thought of like this. The first i items are sorted. Now take the next item and insert it into the part of the data that is already sorted. How does this work? When i is 1, the zeroth item is sorted since only one item is always sorted. So item 1 needs to be either before or after item 1. The inner loop does this. Now the first two items are sorted; we can run the inner loop again to insert the next item at position $i = 2$ into the sorted portion.

What if the data is already sorted? Then the inner loop will always stop on the first test of `data[j] > compare` (this will be false because $j = i - 1$ before the loop, so we are comparing `data[i-1]` with `data[i]`). As soon as the test fails the inner loop can stop because the previous items are sorted. This means that the outer loop will go $n - 1$ times, but the inner loop takes constant time, so the algorithm in this case is $O(n)$.

What of the worst case? The worst-case behaviour is if the inner loop always stops only when $j = 0$. That results in a count of total inner and outer loop iterations described by Equation 3.1, i.e. $O(n^2)$. This will occur if the data is sorted in reverse order, as each item encountered will be smaller than all those previously sorted.

What of the average case? If we know nothing about the distribution of the number of different ways the original data could be ordered, it is reasonable to assume that each time we insert a new item, it will on average be smaller than half the values already sorted. In this case, the inner loop runs half the number of times of the worst case, resulting in time a constant times $\frac{n^2-n}{4}$, which is still $O(n^2)$.

There are several other sorting algorithms with minor differences in efficiency but with similar analysis, best case $O(n)$ when the data is already sorted, worst case $O(n^2)$. Two well-know examples: *bubble sort* and *selection sort*. It is not worth studying these in detail as any requirement for efficiency is better met by $O(n \log n)$ algorithms.

An interesting variation on this kind of algorithm is *shellsort* (named after its inventor, DL Shell) [Shell 1959]. Shellsort works like insertion sort in multiple passes. In each pass, it only compares values a distance d apart, reducing d until $d = 1$. Although the code is more complicated, it can run faster if data is far from sorted as items that are far from their final position move a long distance as a result of one comparison. Analysis of shellsort is very complicated and depends on the exact method of choosing d; some variants run faster than $O(n^2)$ though none as good as $O(n \log n)$ [Plaxton et al. 1992].

The take home message? *Insertionsort is a good method for small data but $O(n^2)$ grows so fast that for n much bigger than 100, a more sophisticated method with lower time complexity is better.*

3.2.2 Efficient methods

The most popular $O(n \log n)$ sort is *quicksort*, invented by Tony Hoare [Hoare 1961]. Quicksort works by:

- *pivot* – choose an element of the array as a basis for reorganising the ordering
- *partition* – reorder as necessary so that all data less than the pivot appears before the pivot
- *recursion* – sort on the two partitions: data before the pivot and data after the pivot

Quicksort works best if the pivot is close to the middle; in the best case, the data is split in half each time, resulting in time complexity $O(n \log n)$. To see why, you need to solve the following *recurrence*, which describes the work done in the algorithm. We call the function $T(n)$ because it approximates the growth rate of run time. It takes n steps to partition n values, and $T(\frac{n}{2})$ describes the amount of work for each of the two recursive sorts of the new partitions:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 \times T(\frac{n}{2}) + n & \text{if } n > 1 \end{cases} \tag{3.2}$$

Intuitively, the total amount of work over all partitions as we divide down remains proportional to n for each pass, even though we divide each partition again at each step. How often can we do another pass over the data? If we divide it evenly, we can do that $\log_2 n$ times. What if the data is already sorted and we choose the first element of the partition as the pivot? Then the algorithm only splits off one sorted item each pass, and its time run is described by Equation 3.1, i.e., it takes time $O(n^2)$. For quicksort, the average case can be shown to be $O(n \log n)$. Since detail of quicksort is simple, a lot of work has gone into tricks to avoid the $O(n^2)$ worst case, rather than on more complex algorithms with worst case $O(n \log n)$.

Solving recurrences can be tricky; if you do a lot of algorithms work, look up the *master theorem*, also known as the *master method*, which provides a cookbook approach to solving recurrences in the form they often come up with algorithm analysis. Since it demonstrates a few useful tricks, I present a proof that this recurrence simplifies to $n \log_2 n + n$ in §3.2.3.

In listing 3.3, I use C-like notation for increment (increase by 1): `i++` means the same as `i = i + 1`, and similar notation for decrement (decrease by 1): `i--`.

This version always choose the first item in the partition as the pivot. To call it: `quicksort (data, 0, N-1)` for N elements (assuming indexing from 0). Another notation detail – I use the Python notation:

```
x, y = y, x
```

for swapping values.

```
quicksort (data, low, high)
   if high > low:
          divide = partition (data, data[low], low, high)
          quicksort (data, divide+1, high)
          quicksort (data, low, divide-1)

partition (data, pivotvalue, low, high) -> int
   loop forever
          while data[high] > pivotvalue
             high--
          while data[low] < pivotvalue
             low++
          if high <= low
             return high
          else
             data[low], data[high] = data[high], data[low]
```

Listing 3.3: Pseudocode: Quicksort

The take home message? *Quicksort is a good general-purpose sorting method; it is not a lot more complex than insertionsort and scales much better for large n with $O(n\log n)$ best and average cases, provided its worst-case behaviour of $O(n^2)$ is avoided.*

It is worth comparing quicksort with *mergesort*, another $O(n\log n)$ sort. Whereas quicksort is $O(n\log n)$ for the best case and average case, it can be $O(n^2)$. Mergesort is *always* $O(n\log n)$, but needs more extra memory and is not as simple so quicksort usually beats it in real examples.

Mergesort works by splitting the data evenly each time (from which it derives its $O(n\log n)$ time), then *merges* the two recursively sorted partitions of the data; recursion continues until there is only 1 item left. Then you pop out of recursion and merge adjacent pairs of values; 1 level up you merge adjacent groups of 4 values, and so on. Dividing the data evenly until there is only one item can be done in $\log_2 n$ steps; merging n items in 2 sorted lists or arrays into one sorted list or array takes $O(n)$ steps. As with quicksort, if we add up the work in all the merges at one level of recursion, there are n items being merged; merges like this happen $\log_2 n$ times, resulting in a total of $O(n \log n)$ work. We can describe the work done with the same recurrence as the quicksort best case (Equation 3.2).

So why is mergesort not the preferred method? Because the constants in the $O(n)$-step merge are bigger than the $O(n)$-step quicksort partition.

Also, I mentioned memory. Quicksort, like any recursive algorithm, needs *stack space* to keep track of the recursion to allow returning to the correct place as each step completes. How deep does the stack go? In the best and average case, $O(\log n)$ extra memory is needed; for the worst case, the stack is $O(n)$ deep. This is *extra memory* over and above the data structures being sorted and is not a constant but a function of n; this is an example pf *space complexity*.

What of mergesort? Any reasonably obvious algorithm to merge two sorted lists, total size n, needs $O(n)$ extra memory. Mergesort needs this in *all* cases – unlike quicksort where the extra memory is much smaller in the best and average cases than in the worst case.

To give you some idea of where the complication in mergesort lies, compare listing 3.4 with that of quicksort (3.3). It looks far simpler. But that's because I separated out the merge code.

```
mergesort (data, low, high)
   if high - low > 1
           divide = (high + low) / 2
           mergesort (data, low, divide)
           mergesort (data, divide, high)
           merge (data, low, divide, high)
```

Listing 3.4: Pseudocode: Mergesort recursion

The merge code is in listing 3.5; that is far more complex and – more importantly – needs a lot of memory since each time we merge, we copy the data to a new data structure, hence the $O(n)$ space complexity. In this listing, I do array slices and ranges in loops with a notation that is explicit about the start and end indices, unlike in Python, where the end of a range is one past the last value. This is consistent with the notation introduced in Listing 1.1.

```
merge (data, low, divide, high)
    templow = data [low..divide-1]
    temphigh = data [divide..high-1]
```

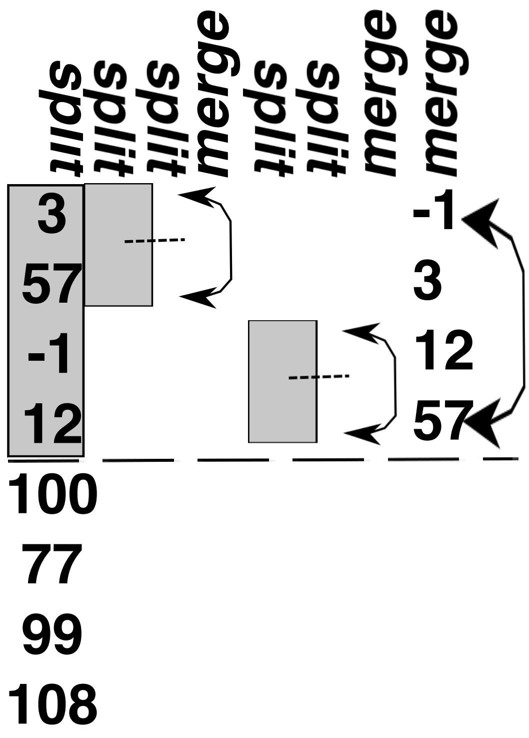


Figure 3.1: Mergesort moves. Showing sorting the first half of the data, stored values only shown when they change. The first 4 elements are split off in the first step and splits continue until partitions have at most one element and are then merged. Only the last merge of elements `[3, 57]` and `[-1, 12]` actually changes the array order since all smaller partitions up to then are sorted.

```
    next_low = next_high = 0
    next_data = low
    low_N = templow.length
    high_N = temphigh.length
    loop forever
      if next_low >= low_N or next_high >= high_N
        break
      if templow[next_low] < temphigh[next_high]
        data[next_data] = templow [next_low]
        next_low++
      else
        data[next_data] = temphigh[next_high]
        next_high++
      next_data++
    for i in 0..low_N-1
      data[next_data] = templow [i]
```

```
        next_data++
      for i in next_high..high_N-1
        data[next_data] = temphigh [i]
        next_data++
```

Listing 3.5: Pseudocode: Merge

A few notes on the code: the first loop finds the smaller next item out of each partition to copy back to the original array; it terminates when one of the two partitions has emptied. The last two loops finish off transferring out of the two partitions when one of them is empty. So one of the loops will have a vacuous range, i.e., it will do zero iterations.

This all works because each partition is sorted prior to merging. Why? Because the recursion continues until it hits a partition with 1 or zero elements, then pops out to the merge code. A partition with 1 or zero elements is by definition sorted. Merge it with its neighbour and the result is a bigger partition that is now sorted. As you pop out of recursion and do a merge, this results in doing so on pairs of partitions sorted by a deeper level of recursion.

Mergesort is a good candidate for switching to a simpler sort when the partitions get small enough as the merge stage of the algorithm is relatively complicated. However, the merge is still $O(n)$ for merging n items, so it is still better than $O(n^2)$ for any reasonably large n.

While quicksort is generally faster, there may be scenarios where you cannot risk quicksort's $O(n^2)$ worst case. Mergesort also adapts better to sorting data too big to fit in memory, where you may need to move part of the data into memory at a time, with the rest in a file. With the trend of flash slowly replacing disk, there has been new work on adapting mergesort to external sorting [Lee et al. 2016].

The take home message? *Mergesort has the benefit of being $O(n \log n)$ even in the worst case, with the downside of requiring $O(n)$ extra memory. Using insertionsort when the partitions are reasonably small is a good trick. Mostly, quicksort is faster but mergesort does have special cases where it is the better choice.*

3.2.3 Proving a recurrence

Equation 3.2 is a relatively easy recurrence to solve. Although we could use the master method, I show here how you can show that it has the following *closed* *form* (a simple equation with no recurrence) for $n = 2^k, k \geq 0$:

$$T(n) = n\log_2 n + n \tag{3.3}$$

I prove by induction on k, i.e., the proof holds for powers of 2 (it is a reasonable approximation for other values). This illustrates a useful trick: doing induction on a new variable, in this case, k, to simplify the proof. The result is inexact for n not a power of 2.

Basis: $k = 0$. For this case, the recurrence directly gives the value 1 and $n\log_2 n + n = 1\log_2 1 + 1 = 0 + 1 = 1$. So the basis case is proved.
Inductive step: assume that $T(n) = n\log_2 n + n$ for some k such that $2^k = n \geq 1$. Now show the result holds for $k+1$, i.e., prove that:

$$\begin{aligned} T(2^{k+1}) &= 2 \times T(\frac{2^{k+1}}{2}) + 2^{k+1} \\ &= 2^{k+1}\log_2 2^{k+1} + 2^{k+1} \end{aligned} \tag{3.4}$$

if it holds for $n = 2^k$, i.e. $T(2^k) = 2^k \log_2 2^k + 2^k$ (*inductive assumption*, IA).

$$\begin{aligned} 2 \times T(\frac{2^{k+1}}{2}) + 2^{k+1} &= 2 \times T(2^k) + 2^{k+1} \\ &= 2 \times (2^k \log_2 2^k + 2^k) + 2^{k+1} \text{ by IA} \\ &= 2 \times (2^k \log_2 2^k) + 2 \times (2^k) + 2^{k+1} \\ &= 2^{k+1}\log_2 2^k + 2^{k+1} + 2^{k+1} \end{aligned}$$

Since $\log_2 2^{k+1} = \log_2(2^k \times 2) = \log_2 2^k + \log_2 2 = \log_2 2^k + 1$, we can write $\log_2 2^k$ as $\log_2 2^{k+1} - 1$:

$$\begin{aligned} 2^{k+1}\log_2 2^k + 2^{k+1} + 2^{k+1} &= 2^{k+1}(\log_2 2^{k+1} - 1) + 2^{k+1} + 2^{k+1} \\ &= 2^{k+1}\log_2 2^{k+1} - 2^{k+1} + 2^{k+1} + 2^{k+1} \\ &= 2^{k+1}\log_2 2^{k+1} + 2^{k+1} \end{aligned} \tag{3.5}$$

So the result holds for 2^{k+1} if it holds for 2^k – the proof is complete since the equations 3.4 and 3.5 are equal.

The take home message? *Solving recurrences in general can be difficult; for recurrences derived from relatively straightforward recursive algorithms, look up the master method.*

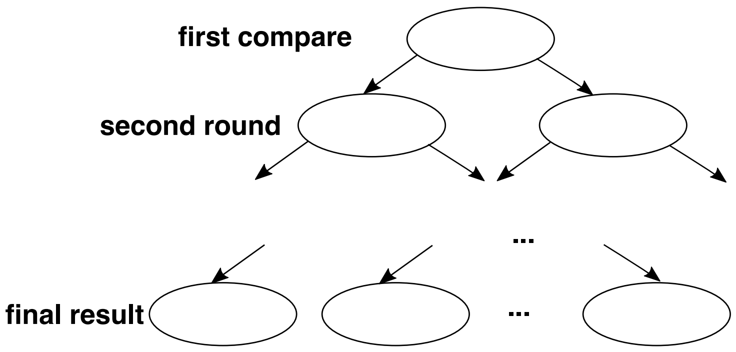


Figure 3.2: Sort decision tree. A record of the order of steps an arbitrary algorithm takes sorting by making a decision after each pairwise comparison. Depending on the outcome of each comparison, you go left or right at each point in the tree to go down one level. Leaves represent all the possible orderings of the original data at least once.

3.2.4 Theoretical limit on sort by comparision

Is $O(n \log n)$ the best we can do for sorting (excluding special cases, like the data is already sorted)? Yes. Mostly. We can prove that the *worst case* and *average case* of any algorithm that makes a decision based on pairwise comparisons cannot have a worst-case analysis better than $O(n \log n)$.

Consider Figure 3.2. This represents all the decisions an algorithm takes sorting by comparing pairs of values. The root and interior nodes represent decisions; depending on the outcome you go left or right down the tree. The picture does not indicate what the decision is – that would depend on the algorithm. The important thing to understand is that the leaves of the tree must number at least as many as the different ways to order the original data. For n values the number of leaves is $n!$ if each reordering is represented exactly once, the smallest number of leaves there could be. If the tree is as compact as possible, it should be balanced, meaning the height of the tree representing the algorithm taking the smallest number of steps possible is to a good approximation $\log_2 n!$

We can approximate $n!$ with Stirling's formula (asymptotically closer to correct as n grows):

$$n! \quad \sim \quad \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{3.6}$$

Consequently (with a little mathematics), $\log n!$ approximates as $n \ln n$ plus lower-order terms, so $O(n \log n)$ is a bound on the worst case of any sort that makes decisions like this. It is also possible to show that this is a bound on the average case.

Does this mean that no sorting algorithm can in general beat $O(n \log n)$? No; to do so you must do something that does not involve making a choice each time you compare values. There are sorting methods such as *radix*, *lexicographic* and *bucket* sorts that put the value directly in the right place, possibly after a few passes over the data. These methods require knowledge of the type of data being sorted to be effective.

The take home message? *Sorting by pairwise comparisons in general cannot do better than $O(n \log n)$. We can do better in special cases and methods that use special knowledge of how the data is represented can also do better.*

3.3 More on Searching

Binary search on a sorted list of array and searching a reasonably balanced binary binary search tree are good options with different trade-offs.

If the data is mostly sorted and not changed often, keeping it in a list or array is reasonable. Adding to a list if you know the right place is quick though if your language – like Python – allows you to index a list as if it's an array, there could be hidden overheads. Adding to an array is expensive as it is usually stored contiguously in memory. If you need to add an element before the end, you need to move everything after it along one position, in the worst and average cases, time $O(n)$. Also, since an array is a fixed-sized structure, adding to it may require reallocating it in new memory and copying *everything* over to the new space, $O(n)$ even in the best case.

A tree has more overhead than an array – you need a left and a right pointer for each node – and is only good if it is kept balanced, possible at the cost of more difficult algorithms for insertion and deletion.

All of these methods can provide $O(\log n)$ search time. Can you do better? There is at least one algorithm that for the average case does better: *interpolation search*. This method, instead of finding the midpoint of each interval, estimates how far along the search key is likely to be by *interpolating* using known values. For example, if the biggest item in an array of 50 elements is 200, the smallest is 10 and the search key is 30, we can estimate how far along it is likely to be assuming a *uniform distribution* of key values. In this case, our estimate (for a

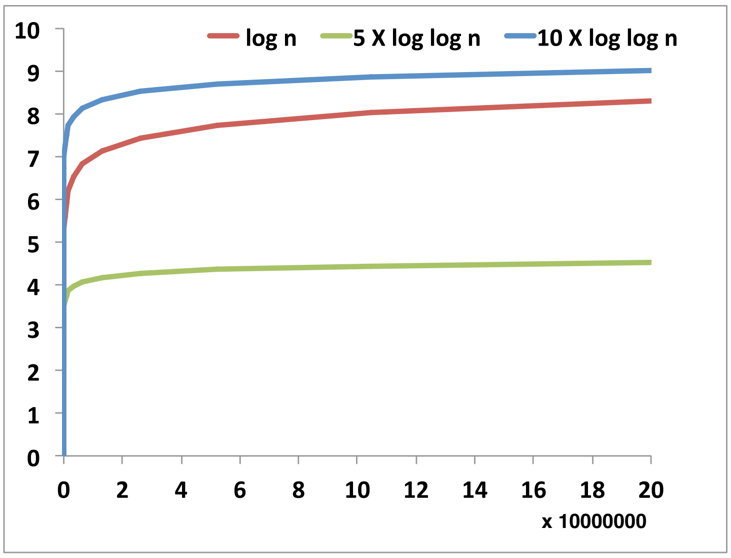


Figure 3.3: $\log N$ *vs.* $\log\log N$. With a small extra overhead per step, $\log\log N$ easily beats $\log N$ even for small N but the actual numbers are not big; with 10 times the overhead, $\log\log N$ is still behind with N=20-million.

partition of an array with indexes in the range $low \dots high$) would be[1]:

$$mid = low + \frac{key - data[low]}{data[high] - data[low]} \times (high - low) \tag{3.7}$$

$$\tag{3.8}$$

The way to read equation 3.7 is: the divide scales the indexes to how far along the key is relative to the minimum and maximum values (the first and last items in the array – since the data is sorted). Now plug in the actual values from our example:

$$\begin{aligned} &= 0 + \frac{30-10}{200-10} \times (49-0) \\ &= \frac{20}{190} \times 49 \\ &\approx 5 \end{aligned}$$

So we would choose 5 as our next index to probe into the array, rather than the midpoint (here, 24). Provided the data is uniformly distributed, this is a

[1]Adapted from https://en.wikipedia.org/wiki/Interpolation_search – where the multiply is done before the divide; doing it that way reduces roundoff error though the formula as expressed here is clearer.

much better guess than probing at the midpoint. How much better? It can be shown that interpolation search on average takes time $O(\log\log n)$ [Perl et al. 1978] – far better than $O(\log n)$ for very large n. *However*, since the algorithm requires significantly more computation than binary search and $O(\log n)$ is small for quite large n, binary search is on the whole preferable. Add to that the fact that interpolation sort's worst case is $O(n)$. In short, interpolation search is an interesting algorithm that could be useful for *really* large data but you need to be sure that the data is reasonably uniformly distributed. If not and you know enough about how it is distributed to perform a reasonably accurate interpolation, you could still get a good result.

Take a look at Figure 3.3. Even for $N = 20,000,000$, $\log N$ is within a factor of two of $5 \times \log\log N$ and a little ahead of $10 \times \log\log N$. That means an $O(\log\log N)$ algorithm has little space for extra complications per step to beat an $O(\log N)$ algorithm except for truly large data.

What then is the best search method? It depends how you process your data. If you read it in once, need to access at least part of it in sorted order and need to search it efficiently, an array is not a bad choice. If you need to update your data frequently and the other requirements are unchanged, a balanced tree is a good choice as insertion and deletion are a lot more efficient than for a tree. If fast searches are your main requirement and a bit of extra memory is not an big issue, a hash table is good.

With a relatively high-level language like Python, you will seldom need to code these things from scratch – just use built-in structures like dictionaries and lists; you need all this if your problem sizes become very large and you do not have pre-existing tools that do the job.

The take home message? *A binary or balanced binary tree search with time $O(\log n)$ is good enough for most purposes. If you are prepared to waste more memory a hash table can do a lot better with $O(1)$ lookups.*

3.4 Further Reading

Quicksort has been widely studied including methods to tweak the details for more speed [Sedgewick 1978] and more recently implementing it on GPUs [Cederman and Tsigas 2009] (using a graphics processing unit for high-speed non-graphical computation is called general-purpose use of GPUs, or GPGPU – that is another whole subject). Knuth [1998] is the most comprehensive reference on sorting and searching – written in what some may consider an archaic style but in a way others may consider comprehensive.

Exercises

1. The quicksort algorithm is an example of *tail recursion* in the second recursive call; it does nothing after the recursion ends. Rewrite the algorithm so the second recursive call is replaced by a loop. What is the value of doing that?

2. Prove by induction that the formula (3.1) for time for an $O(n^2)$ algorithm is correct.

3. Look up how a bucket sort works. Explain why the $O(n \log n)$ construction of Figure 3.2 does not apply and why this sorting method is $O(n)$, no matter how many items we sort. What is the space complexity of the algorithm? Explain.

4. Choose the best data structure and algorithm for each scenario and explain your choice:

   (a) You have data that is captured once then many searches are performed on it and occasionally you need to print out all the data in sorted order.

   (b) You have data that is captured once then many searches are performed on it and you never need to print out all the data in sorted order.

   (c) You have data that is captured a few items at a time and occasional searches are performed on it; you need to print out all the data in sorted order frequently.

5. Rewrite the mergesort algorithm so it switches to insertionsort when partition size is 100 or less.

6. Explain how you would go about implementing interpolation search if the data was not uniformly distributed but you had some idea of the nature of the variability of the distribution of the data.

7. You have a choice of algorithms A and B that respectively have analyses of $O(n^2)$ and $O(n \log n)$. You test them and find A runs faster. Explain how

this could happen and how this could influence your choice of whether to use A or B.

8. Explain why it is not clearly better to use an $O(\log\log n)$ algorithm than an $O(\log n)$ algorithm in all cases.

4 String Matching

STRING MATCHING underlies much of bioinformatics – DNA can be seen as a string from a 4-letter alphabet, RNA from a slightly different 4-letter alphabet and proteins as strings from a 20-letter alphabet. Classic string matching problems often have little resemblance to the problems we solve in bioinformatics because the individual "letters" are not pure abstractions. A string in text processing – a sequence of letters, symbols, punctuation, etc. – can have different meanings depending on context. Are you processing program code or a natural language (one spoken, not created for an artificial purpose, like programming)? Even so, string matching can be reduced to a relatively abstract problem. Does a sequence of a longer string contain an instance of a shorter string? Or, more generally, does a longer string contain a *pattern*, a description that covers a range of possibilities?

In bioinformatics the problem is complicated by three things:

- *data size* – we often need to consider very long strings up to the length of a whole genome and more than one (in the most extreme case, a multi-sequence alignment that includes multiple genomes)

- *differential significance of matches* – in some cases, e.g., amino acids with similar function, a match to individual "letters" can include a requirement to know how substitutable one letter is for another

- *significance of gaps* – in some cases, a gap may not matter; in others it may (e.g., in a coding region)

These and other issues complicate the bioinformatics string matching problem versus the text-processing version. Nonetheless that latter is useful to understand for two reasons: sometimes these classic approaches *are* useful and they are a lot simpler than classic bioinformatics algorithms like Smith-Waterman [Smith and Waterman 1981] and BLAST (Basic Local Alignment Search Tool) [Altschul et al. 1990].

Since Smith-Waterman and BLAST are very complicated, I introduce the basics of string searching in this chapter and look at sequence alignment in the next chapter. Some aspects of string matching apply to contexts other than sequence alignment such as motif discovery and there are occasionally problems where these techniques do apply. So it is worth knowing them both for their potential use and as a scaffold for understanding more complicated algorithms.

Also refer to page 12; the approximate string matching algorithm (*k*-approximate match) may sometimes be of practical use, but also provides a starting point for understanding sequence alignment strategies.

In this chapter, I introduce several exact string match techniques followed by regular expressions, which can be used to match a pattern, not only an exact subsequence.

4.1 Exact Match Algorithms

There is a plethora of interesting algorithms for doing exact-match string searches; I only skim the surface here to indicate the possibilities. The simplest approach is to compare the search string (“pattern”) against the bigger string (“text”) and as soon as there is a mismatch move the pattern along one position and start again. That *naïve* approach works well for small examples but does not take advantage of repetition within the pattern or noticing that one of the characters in the text is not in the pattern, both of which would allow the pattern to move further on a mismatch.

To illustrate the variations, I describe a naïve approach in detail and outline one of the more sophisticated strategies.

To start, it is useful to have a function that detects whether an index is off the end of the text string. Listing 4.1 defines a function `atEnd` that does that.

```
atEnd (str, k) -> boolean
    return k >= length(str)
```

Listing 4.1: Pseudocode: index past end of string

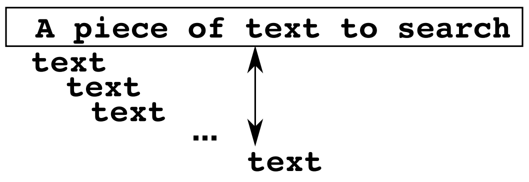


Figure 4.1: Naïve String Match. Each time there is a mismatch, the *pattern* string slides along the *text* string one position. In this case, the pattern being searched for is the string "`text`" The algorithm stops the first time there's a match and returns the position in the text where the match starts – here 11, with the first position counted as 0 – or -1 if there is no match.

4.1.1 Naïve string match

The naïve algorithm, in listing 4.2 and illustrated in Figure 4.1, is based on that of Baase and van Gelder [2000, p 486] and corrects an error in their algorithm. In their version, they check for reaching the end of the text in the loop header; doing this results in missing a match at the very end of the text string. I also simplify the code and make it more obvious but it essentially does the same steps.

```
naiveMatch (pattern, text) -> int
match = -1
i = j = k = 0;
m = length(pattern)
while true
    if k >= m
        match = i
        break
    if atEnd (text, j)
        break
    if text[j] == pattern[k]
        j++; k++
    else
        i++; j = i; k = 0
return match
```

Listing 4.2: Pseudocode: naïve match. Returns the index of the first match of a pattern string in a text string or -1 if not found.

The algorithm returns -1 if there is no match, the index of the match otherwise. In the algorithm, `j` represents where your have got to in the text, `k` how for along you are in the pattern and `i` where the current search started, and `i` is the value returned if the string is found. The line

```
        i++; j = i; k = 0
```

takes `j` back to the position it started the current search plus 1 (after `i++`, it does `j = i`), and the position in the pattern back to the start (`k = 0`).

This algorithm has the merit of being very simple (not much work per loop iteration). However in the worst case (when the pattern is not found) its time complexity is $O(mn)$. To get close to that you would have to construct a pattern of length m that most times *almost* matched before having to give up and sliding it one position. In real natural language texts, that scenario is unlikely; the average time complexity is heavily dependant on use cases.

4.1.2 More advanced string matching

Can we do better? There are algorithms that take advantage of details like repetition within the pattern and skipping over text that contains a character not in the pattern. Some of these tricks do not help us a lot with bioinformatics – for example, skipping over text containing a character not in the pattern is not a great help with DNA searches since a high fraction of patterns contain all four bases. However some of these strategies could be adapted to different approaches so I outline the basic approach of the *Boyer-Moore* algorithm [Boyer and Moore 1977], but present less detail than for the naïve string match and for the k-approximate match algorithms.

Boyer-Moore uses three tricks to reduce the number of comparisons between pattern and text characters:

- *start at end* – start checking the pattern against the string from the end not the start of the pattern
- *skip characters not in the pattern* – is a mismatch occurs on a character not in the pattern, you can skip the pattern forward past the mismatch
- *exploit repeating subsequences* – if a mismatch occurs after a subsequence found earlier in the pattern, skip the pattern forward to align the repeated subsequence

Boyer-Moore was shown early on to have better results than competing approaches, reducing comparisons in English-language text searches to about 25% of the comparisons using rival approaches (depending on length of search pattern,

m) [Smit 1982]. Another approach, Knuth-Morris-Pratt, was shown by this study to be barely better than the naïve approach. The three tricks used by Boyer-Moore all work together. Starting from the back end of the search pattern means you can skip forward as far as possible once you find a mismatch, whichever of the other two strategies you use.

The Boyer-Moore algorithm has a few drawbacks over the naïve algorithm. It takes significantly more setup and it needs data structures such as an array with one entry for every letter of the alphabet Σ (any character that can appear in a text or pattern) that use significant memory. If you are searching a very large text and there is low probability of a match early on, or you are searching for the same pattern multiple times, these disadvantages become less significant. To put the memory size issue into context, an 8-bit character set such as ASCII, commonly used in computers, has 256 different characters, i.e. $|\Sigma| = 256$. An array of 256 integers is not very large and the overhead to set it up is not very large – but if you are searching for a pattern of a few tens of characters at most in a text only a few thousand characters long, this overhead is not worth the effort.

Total time complexity for Boyer-Moore, in addition to scanning for a match, is the time to set up the two data structures ($O(|\Sigma| + m)$) for jumping over characters not in the pattern, for pattern of length m) and $O(m)$ for setting up skipping to the next instance of a subsequence. Extra memory for the two parts of the algorithm is respectively $O(|\Sigma|)$ and $O(m)$. Time for scanning for a match is $O(m+n)$, much better than the naïve algorithm, $O(mn)$.

A lot of work has gone into refining Boyer-Moore; should you need to implement a string matching algorithm, this is a good place to start.

The take home message? *String matching in bioinformatics can be complicated by factors not covered here like reverse complements and amino acids that may have similar functions. Nonetheless the key ideas in Boyer-Moore are useful to understand.*

4.2 Pattern Matching with Regular Expressions

A *regular expression* defines a string in a *regular language*. In formal language theory, languages are classified according to how sophisticated a “machine” can recognise them, i.e., detect whether a string is a member of the language or not. Chomsky [1956] defined 4 classes of language of which regular languages are the simplest.

We need not concern ourselves with the detail of Chomsky's hierarchy: regular expressions are sufficient to define things at the level of complexity of words, but cannot define things we would think of as grammar rules – for that you need a more sophisticated language.

4.2.1 Regular Expressions

There are various ways a regular language can be described; a regular expression is convenient because we can easily create one and interpret it in computer code. There are many variants on regular expressions; they are provided in Python[1], Perl[2] and in Unix scripting in the form of `grep`[3] and `egrep` [4], and in many other programming languages or progamming toolkits.

In general, regular expressions allow you to match:

- *exact* – a specific character
- *set* – one of a selection of characters
- *zero or one* – up to one of the previous item
- *zero or more* – one or none of the previous item
- *one or more* – one or more of the previous item
- *choice* – select between subsequences
- *repetition* – repetition a specific number of times of the previous item

To make this concrete, I introduce a simplified notation to use for examples, noting that grep symbols as defined here cannot be matched unless they are *escaped* as noted below this list:

- *exact* – any character not used as a special symbol, or a special symbol preceded by "\"
- *any character* – a full stop: "."

[1]`https://docs.python.org/2/library/re.html`
[2]`https://perldoc.perl.org/perlre.html`
[3]`https://www.gnu.org/software/findutils/manual/html_node/find_html/grep-regular-expression-syntax.html`
[4]`https://www.gnu.org/software/findutils/manual/html_node/find_html/egrep-regular-expression-syntax.html`

- *set* – characters enclosed in "`[ ]`": select only one of them; we can give ranges (such as "`[a-z0-9]`", meaning all lowercase letters and all digits)
- *zero or one* – any character or group followed by "?"
- *zero or more* – any character or group followed by "∗"
- *one or more* – any character or group followed by "+"
- *an exact count* – $\mathtt{c}^{\mathtt{n}}$ means n of any character or group `c`
- *choice* – two characters or groups separated by "`|`"
- *group* – one or more characters or grep symbols enclosed in "`( )`"

The "repetition" option in the list of features can be accommodated by repeating a group, enclosed in parentheses. All the "special characters" (grep symbols), used to indicate expression syntax, can be turned into a character to recognize using the *escape character*, "\" – and that includes "\" itself, e.g. "`\\A\\.*`" would match any of the following (plus any other example that started the same way):

- "`\A\`"
- "`\A\Docs`"

In this case, the double-"\\" in both cases stands for a single "\", the "`.`" matches anything and the "∗" says as many as you like. Let's say now we want to only allow letters of the alphabet rather than anything for the variable part of the pattern. It becomes: "`\\A\\[a-zA-Z]*`".

4.2.2 Finite state machine

A finite state machine (FSM) can *recognize* a regular language (i.e., correctly tell you if a string is a member of that language or not). You can represent a finite state machine (FSM; also called finite automaton) as a digraph with labelled transitions. An FSM in effect represents a regular expression (and hence the regular language it represents). An FSM has a *start state* where recognition begins and each state can have zero or more transitions, each labelled by what must match to go that path. If you reach the end of a string on a *final state* marked by a double circle, you have recognized something that is defined by the language.

Figure 4.2 illustrates a finite state machine (FSM) that recognises the language described above.

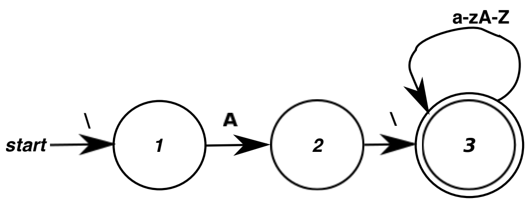


Figure 4.2: Finite State Machine. It recognizes a regular language that starts with a backslash, a letter "`A`", a backslash then zero or more letters of the alphabet (capital or lowercase).

An FSM can also be represented as a table with one row for each state and one column for each letter in the alphabet. For each state, the transitions out of that state are given in the column for the letter (or other character, like a backslash as in the example); if there is no transition, encountering that character is an error. Table 4.1 represents the FSM of Figure 4.2 in tablular form. You start at the start state; any transitions labelled "error" (implicitly also any characters not listed result in an error) are incorrect. So from the start state, only a backslash is an allowed input taking you to state 1. Once there, only an `A` is allowed, taking you to state 2. From there, only a backslash is allowed, taking you to state 3. At state 3, you can only keep going to state 3 if any of the inputs `A`, `a`–`z`, `B`–`z` occur. But you can stop any time in state 3, since it's a final state. Here, since there is only one transition that uses the whole alphabet, I can put that in one column (minus the letter "`A`", which does appear in a separate transition).

4.2.3 Regular language limitations

What are the limits of regular languages? They cannot count in more than one place at once. For example, they cannot match parentheses. A language L with any

Table 4.1: Finite State Machine Table. Each row represents what state to go to next given the current state and the next character.

	character			
state	\	**A**	**a–z,B–Z**	**final?**
start	***1***	**error**	**error**	**no**
1	**error**	***2***	**error**	**no**
2	***3***	**error**	**error**	**no**
3	**error**	***3***	***3***	***yes***

number of open parentheses, any number of lowercase letters, the same number of close parenthesis as opened the string, defined by (note the need to escape the parenthesis characters with “\”, since parentheses are used to group characters in our notation):

$$L = \texttt{\textbackslash(}^{\texttt{n}}\texttt{[a-z]}*\texttt{\textbackslash)}^{\texttt{n}} \tag{4.1}$$

is not regular. Why? Prove by contradiction. Assume you can create a finite state machine that recognizes $\texttt{\textbackslash(}^{\texttt{k}}\texttt{.}*\texttt{\textbackslash)}^{\texttt{k}}$ for some k. Observe that if $\texttt{\textbackslash(}^{\texttt{k}}\texttt{[a-z]}*\texttt{\textbackslash)}^{\texttt{k}}$ is recognized, the finite state machine that recognizes this can be split into the part that gets as far as $\texttt{\textbackslash(}^{\texttt{k}}\texttt{[a-z]}*$ and the other part that recognizes $\texttt{\textbackslash)}^{\texttt{k}}$, so nothing can stop if from also recognizing $\texttt{\textbackslash(}^{\texttt{k}}\texttt{[a-z]}*\texttt{\textbackslash)}^{\texttt{k+1}}$.

This is a very informal demonstration; the more formal form of proof is called the *pumping lemma for regular languages*. The key idea is that there has to be a maximum size of string you can recognize in a finite number of states before you have a loop. Once you have a loop, you cannot force that loop to go the same number of times as any other loop in that finite state machine, so any construct that requires counting in two different places cannot be recognized (by this, we generally mean, without recognizing other strings that fall outside the language).

Despite this limitation, regular expressions and their recognizers are very powerful, hence their availability in many programming languages (either as a construct or as part of a library or module). A note of caution though: a complex regular expression takes significant computation time to construct the recognizer. if you use simple examples or if you repeatedly use the same example (either to search a very large text, or to search for the same pattern), they are very useful.

The take home message? *Regular expressions are a powerful notation for constructing searches that are more complex than a fixed pattern string. They have some setup cost, meaning they are most useful for small regular expressions or for repeated use once the internal representation of the FSM is set up.*

4.3 Further Reading

If you want to learn more about finite automata, there are good books about formal languages [Hopcroft et al. 2013] and compilers [Ullman et al. 2014]. String search is a complex subject with many special cases. Review the original Boyer-Moore paper [Boyer and Moore 1977] and a selection of work that cites it.

Exercises

1. Work through the example in Figure 4.1 to demonstrate how the naïve match algorithm of Listing 4.2 works.

2. Apply the trick of the Booyer-Moore algorithm of scanning the pattern backwards and shifting the pattern right past any character not in the text. How much faster do you go with the Figure 4.1 example?

3. Go to Google Scholar and look for bioinformatics work that cites the original Boore-Moore algorithm. Restrict the cited works to "Since 2013". Comment on what you find.

4. If you are searching for a single pattern of length 5 in a text of length 10,000, would you use the naïve or Boyer-Moore approach?

5. If you are searching multiple times for the same pattern of length 5 in a text of length 10,000, would you use the naïve or Boyer-Moore approach? Is your answer any different to the previous? Why?

6. You need to search for a string that contains a prefix "`AT`", any number of letters then "`TA`". Which of the methods covered in this chapter would you use? Why?

7. Compare implementations of regular expressions in Python and the command-line utilties `grep` and `egrep`. List differences in notation and capability.

8. A *nondeterministic finite automaton* (NFA) can have more than one transition out of the same state for a given input. Expain why it is a good thing that it is always possible to convert an NFA into a *deterministic finite automaton* (DFA).

9. Look up algorithms for constructing a finite automaton from regular expression. Is a regular expression something you should use when a simpler search is possible? Explain.

5 Sequence Alignment

SEQUENCE ALIGNMENT is one of the most used techniques in bioinformatics and has many applications. The general form of the problem is: given two or more sequences, find the best alignment of all the sequences or selected subsequences.

The simplest approach is something along the lines of the approximate string matching algorithm of page 11. That approach is $O(mn)$ where m and n are lengths of the two strings being compared so it could be adapted to comparing short sequences, e.g., smaller proteins or clearly defined parts of genomes. However for many interesting cases of sequence alignment such as whole genome alignments, an approach that is a lot faster than this is required. In practice, practical approaches use *heuristics* and provide an approximate alignment, usually good enough for practical purposes.

Essential properties of an alignment are much as in the approximate string matching algorithm. At each position, the nucleotide or amino acid in each sequence being compared can:

- match
- mismatch but in the right position: a *substitution*
- mismatch because there is an *insertion*
- mismatch because there is an *deletion*

The latter two cases are often collectively termed an *indel*.

In this chapter, I introduce an attempt at doing exact alignments (Smith-Waterman) and the widely-used BLAST algorithm that is more scalable at the cost of being approximate.

Heads up: *This chapter is a placeholder – the focus of the course developed differently than planned based on class feedback.*

5.1 Further Reading

There is a proof that it is possible to find a subsequence of a sequence of length n that has at most k mismatches to a shorter string of length m in time $O(n \log k + \frac{nk^3 \log k}{m})$ [Amir et al. 2004]. This looks pretty exciting, but you have to remember that the general sequence alignment problem does not start from knowing what k is. Each time you repeat the algorithm for another value of k, you add another multiple of n to the run time.

6 Case Studies

I CONSIDER TWO EXAMPLES IN THIS CHAPTER. One uses the regular expression concept developed in Chapter 4, the other illustrates how to go from a concept to a finished program in manageable steps. Both relate to DNA analysis.

Also please take a look at the figure on page ii for a generic plan for designing algorithms and ending up with a finished program.

The first example, DREME, finds motifs representing transcription factor binding sites. The second, CentriMo, does central motif enrichment analysis. Both illustrate how it is possible to develop an idea with tools and languages that are quick and easy to program, moving on to more efficient implementations once the ideas are worked through.

6.1 Motif Finding with Regular Expressions

I start out by describing a motif and one way of representing one. Motifs are an important tool for analysis of DNA-protein interactions and protein characteristics. I then explain a method for finding motifs using regular expressions that illustrates how some of the material in earlier chapters is useful for developing new tools and algorithms in bioinformatics.

6.1.1 Motifs

A *motif* is a pattern found in DNA or protein. It may represent any feature of interest such as a repeated pattern in a protein, a pattern common to more than one protein or a transcription factor binding site (TFBS) in DNA. Motifs, representing binding of a TF (TF), can be represented in various ways that represent the probability of a particular base (or amino acid in other applications of motifs)

```
0.338561 0.018681 0.235701 0.407057
0.020276 0.002074 0.976267 0.001382
0.003223 0.002993 0.990792 0.002993
0.003221 0.008282 0.984817 0.003681
0.063693 0.441941 0.002529 0.491837
0.005064 0.003453 0.983656 0.007827
0.009671 0.018420 0.501727 0.470182
0.060872 0.010606 0.899700 0.028822
0.028400 0.030016 0.874856 0.066728
0.058742 0.660962 0.064755 0.215541
```

Figure 6.1: MEME motif matrix. Each column represents a base in the order A, C, G, T for DNA; protein motifs are also possible. The MEME format in addition names the motif and has additional information to aid in processing such as the size of the alphabet.

at a particular point in a sequence. This kind of representation is called a *position weight matrix* or PWM – sometimes also a *position-specific score matrix*, PSSM.

For purpose of illustration I use MEME-format motifs, which – in minimal form with other file contents stripped out – are a matrix representing a probability value for each letter in the alphabet under consideration (DNA or protein) for each position. MEME is one of the older motif finders [Bailey and Elkan 1994] and still in wide use. In MEME format, each column represents one letter and each row one position within a site.

Figure 6.1 contains an example of the matrix for a MEME motif. A motif in this form is an example of a *position weight matrix* (PWM); there are many other formats and methods for scoring each position. This example is Klf4 from JASPAR CORE database [Mathelier et al. 2013].

Since visualizing a matrix is difficult – particularly as there are differing standards as to what a PWM contains – it is useful to have a graphical representation of a motif. A *sequence logo* ("logo" for brevity) illustrates the contents of a PWM by stacking the letters in each position, from lowest (bottom) to highest probability. The letters are scaled for probability and the height of the stack at each position is scaled for *information content* (IC) of the position [Schneider and Stephens 1990]. Intuitively, the more uncertainty as to the specificity, the lower the information content so, e.g., if each letter has equal probability, the information content is zero. For a 4-letter alphabet, the maximum IC at a position is 2 bits of information, if we know precisely which letter is represented. So IC at position i is defined as

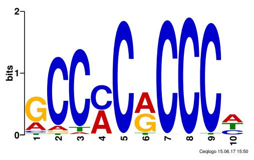


Figure 6.2: Logo representation of a motif. Klf4 from JASPAR CORE. Overall height of letters is the information content of that position; height of each letter is its proportion of the total information.

follows (for base $j = A, C, G, T$, frequency $f(j,i) > 0$ of base j at position i):

$$IC(i) = 2 + \sum_{j=A}^{T} f(j,i) \times \log_2 f(j,i) \tag{6.1}$$

If $f(j,i) = 0$, we define the contribution of base j at position i as zero to avoid a singularity. With this definition, equal probabilities ($f(j,i) = 0.25$ for all j at position i) results in zero IC; if $f(j,i) = 1$ for one base at position i and all others are 0, IC is 2.

The formula in general is

$$IC(i) = \log |\Sigma| + \sum_{\Sigma} f(j,i) \times \log_2 f(j,i) \tag{6.2}$$

where Σ is the alphabet of interest (not to be confused with the summation sign) – i.e., for proteins, the first term changes from $\log_2 4 = 2$ to $\log_2 20 \approx 4.322$, and the probabilities are summed over 20 amino acids instead of 4 bases.

Figure 6.2 illustrates the same Klf4 motif as a logo. Note that the logo is for the motif on the opposite strand, hence the letter probabilities do not look right unless you read the logo as its reverse complement (swap the roles of A and T, C and G and read it backwards).

The *de novo* motif discovery problem requires finding these patterns in sequences of interest and identifying those of most statistical significance. There are many methods for doing this.

6.1.2 The DREME motif finder

DREME (Discriminative Regular Expression Motif Elicitation) is a motif finder designed to find all short core transcription factor binding motifs in a large DNA data set. This has become a requirement with the growing popularity of

Chromatin ImmunoPrecipitation followed by high-throughput sequencing (ChIP-seq) to identify transcription factor binding sites. ChIP-seq generates large data sets, sometimes containing tens of thousands of sites.

DREME starts from finding motifs as simplified regular expressions and converts these to MEME format PWMs [Bailey 2011] after working out their significance.

The starting point is counting all k-mers (DNA sequence k letters or bases long) for $k \in \{3, \ldots, 8\}$ in a *positive* sequence set and calculating their significance versus their count in a *negative* sequence set. The negative set constitutes a background model; you can supply a set of sequences that are known not to contain binding sites or let DREME shuffle the positive sequences to form the negative set. The most significant 100 k-mers are then used to seed the model, and form the basis for finding regular expressions that generalise these most significant words. Each iteration of the program considers one letter replaced by a *wildcard* (match any) to estimate how much variability in the word is the best model (can be found with highest significance, as measured by prevalence in the positive versus negative set).

Since scanning a regular expression over a string is efficient, $O(n)$ for scanning the whole string if matches are not allowed to overlap and the program runs a fixed number of iterations, the total time complexity is $O(n)$ and scales up well to large data sets.

DREME was originally written in Python for speed of coding; the current version is written in C for speed of execution.

6.1.3 CentriMo motif central enrichment analysis

Finding motifs from scratch is a useful exercise but it can also be useful to check how well known motifs are represented in a sequence set. Doing so is called *motif enrichment analysis* (MEA). The general MEA problems is: given a database of motifs, work out the relative representation of those motifs in a set of possible binding sites. A motif that has a high likelihood fit to a large number of sites is relatively highly enriched; as the quality of fit to sites and the number of sites with any match to the motif reduces, it is less enriched.

This generic description does not cover details like *how* a motif is matched to sites or when a match is or is not counted. There are various scoring methods. One of the most common is the *log likelihood ratio*. The simplest model of likelihood is to place the matrix containing probabilities for each letter over a site and multiply the probabilities according to which letter is present divided by the *background probability*.

Likelihood is defined for a motif of length m by

$$L = \prod_{i=1}^{m} \frac{PWM_{i,b}}{P_{back}(b)} \tag{6.3}$$

where b is a base, $PWM_{i,b}$ is the entry at position i for base b and $P_{back}(b)$ is the background probability for base b. Taking a log of likelihood is a way of scaling and also makes it possible to do arithmetic with additions instead of multiplications at the cost of having to introduce a small value greater than 0 as the minimum to avoid $\log 0$. The detail is not important here. The important thing is that we can score a motif, choose a threshold above which to claim a match and compare scores.

MEA methods may in some cases simply count how often a match occurs above a threshold, rather than use the score. Others may take the maximum score in each sequence.

The idea

CentriMo [Bailey and Machanick 2012] arose from the fact that ChIP-seq is relatively good at locating binding sites so the true binding motif for a TF should be biased towards the centre of sequences found in a ChIP-seq experiment. So the approach is to take a motif database, one motif at a time, and find the best match to that motif in every sequence in the ChIP-seq data set. Then, with a preferred site in each sequence (except if there is no match good enough to use), it is possible to score the relative abundance in sites within a central region versus a background probability that the sites are uniformly distributed.

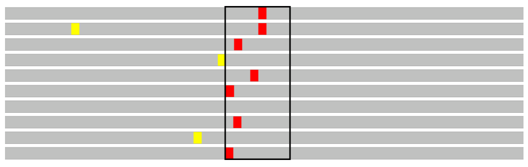


Figure 6.3: Central Motif Enrichment Analysis Concept. CentriMo counts how often the motif is found within a central window within a set of sequences (each grey line represents a sequence) then works out the binomial p-value, indicating the probability that this differs significantly from a uniform distribution of sites.

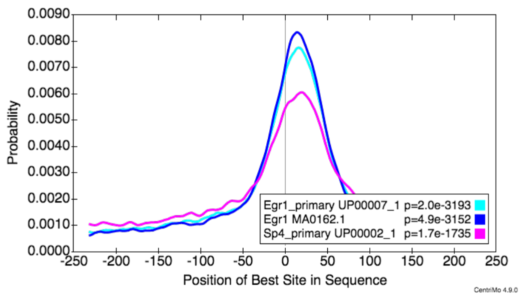


Figure 6.4: Centrimo output. The most centred motif enrichment graph is most likely to represent the true binding motif.

Figure 6.3 illustrates the essential concept of CentriMo (*Centr*al*i*ty of *Mo*tifs). The program scores different widths of central region and chooses the one with the lowest binomial p-value. The probability calculation assumes that the background probability is uniform e.g., if the central region is a quarter of the width of the sequences, the probability of placing the best match to the motif in that region should be 0.25. That value is compared with the actual fraction in the centre to obtain a binomial p-value.

A highly centred sharply defined distribution is likely to represent the motif of the TF that was used to produce the sequence set. Sometimes though binding is indirect or requires cofactors, resulting in less clearly defined central enrichment – or no central enrichment at all. Cofactors may sometimes result in multiple distributions not quite centred, indicating binding sites of other TFs that often bind near the TF of interest. Indirect binding may result in no clear pattern as the actual TFs binding to the DNA may not be the TF of interest. You can see an example output of CentriMo in Figure 6.4.

The prototype phase

Tim Bailey approached me with the idea behind CentriMo. To do a preliminary evaluation, I put something together quickly using existing tools.

The FIMO program [Grant et al. 2011] already has a feature to score motifs against sequences and can produce plain-text output from which I could extract the required best site and score for each motif for each sequence. From that, I could calculate the binomial p-value. All of this I did with a combination of running existing programs, piping output to scripts and piping the numbers I needed to a script written in Perl, a language popular at the time for bioinformatics programming. Perl has features like hash tables and quick and easy string manipulations.

In its original form, the program was fairly crude but showed that the model proposed for central enrichment analysis worked well enough to invest more time in it.

All operations were at worst $O(n)$ for n bases counted across all sequences, which meant that even this crude prototype with a language designed for speedy coding not speedy running ran reasonably fast.

Final implementation

In the next stage, I reimplemented it in C, drawing on the code that already existed for the implementation of FIMO. The program has subsequently been further refined and is part of the MEME suite, including an easy to use web version called MEME-ChIP that integrates CentriMo into a variety of other tools.

The final version is a lot different than the original. The fact that all operations on the original data were at worst $O(n)$ meant that converting to C resulted in a nice speedup – the original work ensured that this would be worthwhile.

6.2 Further Reading

CentriMo has been significantly refined since I worked on it including the addition of *differential mode* that compares central enrichment on two sets of sequences and *non-centred enrichment* that can find enrichment anywhere along a sequence set [Lesluyes et al. 2014]. You can find out more about the MEME-ChIP toolchain that includes DREME and Centrimo in Machanick and Bailey [2011].

Exercises

1. A PWM (with many variants on the actual numbers stored) is the default way to represent a motif. Could you use a regular expression instead?

2. If you were coding something like DREME in Python, what data structure would you use to keep counts of k-mers?

3. Take the given motif for KLF4 (Figure 6.1) and work out its likelihood score for:

   (a) AGGGTGTGGC

   (b) GCCACACCCT
   (c) Are the results as expected?

4. If a CentriMo run does not show anything as centrally enriched, how would you interpret that?

5. You have Python code to compute a range of statistical functions including a binominal p-value and you can run the FIMO program to obtain a score of a motif in each sequence of a set of binding sites. Describe how you would combine these things to produce something similar to CentriMo.

6. The final version of CentriMo that I implemented before I left the project, is implemented in C. I had access to the C code for FIMO, which I was able to reuse. Discuss the reasons I might have had for reimplementing in C.

7. The final version of DREME is written in C and uses a red-black tree to store frequency counts rather than a hash table. Discuss why this design decision could have been taken.

8. The implementation of DREME uses a very simplified form of regular expression, only allowing one character at a time to be replaced by a “wildcard”. Discuss whether this could be implemented without a full regular expression feature in a programming language.

9. Check out the generic design plan in the picture in the Preface (page ii). How does this design plan fit the way CentriMo and DREME were developed?

References

Adelson-Velsky, G. M. and Landis, E. M. (1962). An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266. In Russian: English translation by Myron J. Ricci in *Soviet Math. Doklady*, 3:1259–1263, 1962.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular biology*, 215(3):403–410.

Amir, A., Lewenstein, M., and Porat, E. (2004). Faster algorithms for string matching with *k* mismatches. *Journal of Algorithms*, 50(2):257–275.

Baase, S. and van Gelder, A. (2000). *Computer algorithms: introduction to design and analysis*. Addison-Wesley, Reading, MA, 3rd edition.

Bailey, T. L. (2011). DREME: motif discovery in transcription factor ChIP-seq data. *Bioinformatics*, 27(12):1653–1659.

Bailey, T. L. and Elkan, C. (1994). Fitting a mixture model by expectation maximization to discover motifs in bipolymers. In *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, pages 28–36. AAAI Press, Menlo Park, California.

Bailey, T. L. and Machanick, P. (2012). Inferring direct DNA binding from ChIP-seq. *Nucleic Acids Research*, 40(17):e128–e128.

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772.

Cederman, D. and Tsigas, P. (2009). GPU-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA)*, 14:4.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition.

Coron, J.-S., Dodis, Y., Malinaud, C., and Puniya, P. (2005). Merkle-Damgård revisited: How to construct a hash function. In *Annual International Cryptology Conference*, pages 430–448. Springer.

Elloumi, M. and Zomaya, A. Y. (2011). *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications*. John Wiley & Sons, Hoboken, NJ.

Grant, C. E., Bailey, T. L., and Noble, W. S. (2011). FIMO: scanning for occurrences of a given motif. *Bioinformatics*, 27(7):1017–1018.

Guibas, L. J. and Sedgewick, R. (1978). A dichromatic framework for balanced trees. In *Proc. 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE.

Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321–321.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2013). *Automata Theory, Languages, and Computation*. Pearson Education, Harlow, 3rd edition.

Knuth, D. E. (1998). *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Upper Saddle River, NJ, 2nd edition.

Lee, J., Roh, H., and Park, S. (2016). External mergesort for flash-based solid state drives. *IEEE Transactions on Computers*, 65(5):1518–1527.

Lesluyes, T., Johnson, J., Machanick, P., and Bailey, T. L. (2014). Differential motif enrichment analysis of paired ChIP-seq experiments. *BMC Genomics*, 15(1):752.

Machanick, P. and Bailey, T. L. (2011). MEME-ChIP: motif analysis of large DNA datasets. *Bioinformatics*, 27(12):1696.

Mathelier, A., Zhao, X., Zhang, A. W., Parcy, F., Worsley-Hunt, R., Arenillas, D. J., Buchman, S., Chen, C.-y., Chou, A., Ienasescu, H., et al. (2013). JASPAR 2014: an extensively expanded and updated open-access database of transcription factor binding profiles. *Nucleic acids research*, page gkt997.

Maurer, W. D. and Lewis, T. G. (1975). Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19.

Perl, Y., Itai, A., and Avni, H. (1978). Interpolation search – a log log$n$ search. *Communications of the ACM*, 21(7):550–553.

Plaxton, C. G., Poonen, B., and Suel, T. (1992). Improved lower bounds for shellsort. In *Proceedings of 33rd Annual Symposium on Foundations of Computer Science*, pages 226–235.

Rizk, G., Lavenier, D., and Chikhi, R. (2013). DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653.

Schneider, T. D. and Stephens, R. M. (1990). Sequence logos: a new way to display consensus sequences. *Nucleic Acids Research*, 18(20):6097–6100.

Sedgewick, R. (1978). Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857.

Shell, D. L. (1959). A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32.

Smit, G. d. V. (1982). A comparison of three string matching algorithms. *Software: Practice and Experience*, 12(1):57–66.

Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197.

Turing, A. M. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265.

Ullman, J. D., Aho, A. V., and Sethi, R. (2014). *Compilers Principles, Techniques, and Tools*. Pearson Education, Harlow, 2nd edition.