NICOLAS WU

ALGORITHM DESIGN & ANALYSIS

Contents

Foreword

These lecture notes were produced to accompany the second-year Algorithm Design & Analysis course at Imperial College London.

1
Introduction

> It is not only the violin that shapes the violinist, we are all shaped by the tools we train ourselves to use, and in this respect programming languages have a devious influence: they shape our thinking habits.
>
> E. Dijkstra, 2001

An algorithm is a method for computing a desired result. At the heart of algorithms is the aim to produce some systematic, even mathematical, means of producing a solution to a given problem.

The word *algorithm* itself comes from the name of a 9th Century mathematician from Persia, Muḥammad ibn Mūsā al-Khwārizmī who was immortalised by his work on algebra,where amongst other things he showed how to solve quadratic equations by completing the square.

This course is about the design, analysis, and implementation of algorithms. The goal, broadly, will be to understand and recognise a range of algorithmic techniques in order to solve problems. Unlike in most presentations the algorithms themselves will not be presented in pseudo-code, but rather, as implementations that can be executed.

To this end, the code will be in Haskell, a functional language that excels at expressing datastructures and recursive functions in a succinct and clear manner. Although Haskell will be the vehicle to describe the ideas in this course, the lessons learnt about algorithm design and analysis are equally applicable in other languages.

There is something fishy about algorithms. The word algorithm is from the French algorithme, from the old French algorisme, from the Medieval Latin algorismus, from the Arabic al-Khwārizmī, from the Persian Xwârazm, which means cooked fish (from khwar and razm). However, Xwârazm might alternatively be translated kh(w)ar, "low" and zam "land", since it is the lowest region in Persia. Now, the word beneath comes from *be-* and *neath*, where neath means low. This is also the same word as in the *Netherlands*, which is also, therefore, another word for algorithm. Finally, in the Netherlands is home to many *dykes*, which are trenches which people *dig*. In Germanic languages, the suffix *-stra* means dweller, so these people could be called dykestras. As it happens Edsger Dijkstra wrote a famous algorithm that we will study.

1.1 Fundamentals First

Some fundamentals are required before the study of more advanced algorithms can be approached. This section introduces the simple example of sorting as a means of fixing notation and refreshing some basics that should already be familiar.

Given an integer x and sorted list of integers ys, the *insertion problem* is to produce the list containing the elements $x : ys$ in sorted order. One algorithm would be to simply sort this list, that is, *sort* $(x : xs)$, but this seems to be overkill, in that it does not exploit the fact that *xs* is already sorted.

A more sensible approach is to traverse the list *ys* element by element until the correct location for x has been found. Then, x can be inserted into the right place and the algorithm is complete. This idea is implemented in the *insert* function, defined as follows:

```
insert :: Int → [Int] → [Int]
insert x [ ] = [x]
insert x (y : ys)
  | x ⩽ y      = x : y : ys
  | otherwise = y : insert x ys
```

In the base case, the list is empty, so all that is needed is the list containing the element x. In the recursive case, $y : ys$ is a sorted list. If $x \leqslant y$, then x belongs at the beginning of this list. Otherwise, the element y must come first, followed by the result of inserting x into the remaining list *ys*.

Understanding the computational complexity of the *insert* function involves counting the number of *call steps* that are required to process the input. A call step is counted each time an application of a non-primitive function is reduced. For example, here is a trace of how many call steps are required to evaluate *insert* $4\ [1,3,6,7]$, assuming that each comparison $(\leqslant)$ is primitive:

```
   insert 4 [1,3,6,7,9]
⇝    { definition of insert }
   1 : insert 4 [3,6,7,9]
⇝    { definition of insert }
   1 : 3 : insert 4 [6,7,9]
⇝    { definition of insert }
   1 : 3 : 4 : 6 : [7,9]
```

Note that $[1,2]$ is syntactic sugar for $1 : 2 : [\,]$, and there is no reduction between these terms.

This took 3 call steps, each using the definition of *insert*.

Accurately counting steps by evaluation is a tedious task, and it is often enough to work instead with a more convenient *recurrence relation* that abstracts and approximates this count by focusing on how the size of input data affects the number of reductions. Solving the recurrence relation will give a closed-form solution that is easier to compute and understand.

An accurate step counter of equation reductions isn't that useful in practice: the binary produced by a modern compiler that corresponds to this code will not usually perform such reductions.

The recurrence relation for the evaluation of *insert* $x\ xs$ in the worst case will be given by $T_{insert}(n)$, where the measure of input size is the length of the list *xs*, given by $n = length\ xs$. Defining this relation is achieved by looking at the cost of each clause in the definition of *insert*. In the base case, $xs = [\,]$ and so $n = 0$, only one reduction is required. In the recursive case $n > 0$, and at worst *insert* is invoked again with a list of size $n - 1$.

$$T_{insert}\ 0 = 1$$
$$T_{insert}\ n = 1 + T_{insert}\ (n-1)$$

This recursion equation is then solved to give a closed-form solution, where recursion in the recurrence relation is removed. A strategy for discovering the closed-form solution is to simply unroll the definition and look for patterns.

$$\begin{aligned}
T_{insert}\ n &= 1 + T_{insert}\ (n-1) \\
&= 1 + 1 + T_{insert}\ (n-2) \\
&= ... \\
&= 1 + 1 + ... + T_{insert}\ (n-n) \\
&= 1 + n
\end{aligned}$$

Thus it takes approximately n steps to compute *insert x xs*, where $n = \mathit{length}\ \ \mathit{xs}$, a fact that is captured by saying that *insert* has $O(n)$ complexity. This is an example of *asymptotic notation*, which formalises the idea that it is useful to be able to provide an approximate measure of how long it takes for an algorithm to complete given some input size.

Here is another problem. Given a list of values, the *sorting problem* must produce the list containing those values in ascending order. One way to solve this problem is to use the *insertion sort* algorithm, which constructs a sorted list by taking each element in the input and inserting it into an initially empty sorted list.

$$\begin{aligned}
&\mathit{isort} :: [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{isort}\ [\,] \qquad\ \ = [\,] \\
&\mathit{isort}\ (x : \mathit{xs}) = \mathit{insert}\ x\ (\mathit{isort}\ \mathit{xs})
\end{aligned}$$

Structural induction on the input list leads to a natural algorithm. If the list is empty then it is already sorted. Otherwise, an element x is inserted into the result of sorting xs using the *insert* function.

Here is a small example of this function evaluating fully.

With some luck, this might the last time an evaluation this long is written.

$$\begin{aligned}
&\quad \mathit{isort}\ [3,1,2] \\
&\rightsquigarrow \quad \{ \text{definition of } \mathit{isort} \} \\
&\quad \mathit{insert}\ 3\ (\mathit{isort}\ [1,2]) \\
&\rightsquigarrow \quad \{ \text{definition of } \mathit{isort} \} \\
&\quad \mathit{insert}\ 3\ (\mathit{insert}\ 1\ (\mathit{isort}\ [2]) \\
&\rightsquigarrow \quad \{ \text{definition of } \mathit{isort} \} \\
&\quad \mathit{insert}\ 3\ (\mathit{insert}\ 1\ (\mathit{insert}\ 2\ (\mathit{isort}\ [\,]))) \\
&\rightsquigarrow \quad \{ \text{definition of } \mathit{isort} \} \\
&\quad \mathit{insert}\ 3\ (\mathit{insert}\ 1\ (\mathit{insert}\ 2\ [\,]) \\
&\rightsquigarrow \quad \{ \text{definition of } \mathit{insert} \} \\
&\quad \mathit{insert}\ 3\ (\mathit{insert}\ 1\ [2]) \\
&\rightsquigarrow \quad \{ \text{definition of } \mathit{insert} \} \\
&\quad \mathit{insert}\ 3\ [1,2] \\
&\rightsquigarrow \quad \{ \text{definition of } \mathit{insert} \} \\
&\quad 1 : \mathit{insert}\ 3\ [2] \\
&\rightsquigarrow \quad \{ \text{definition of } \mathit{insert} \} \\
&\quad 1 : 2 : \mathit{insert}\ 3\ [\,] \\
&\rightsquigarrow \quad \{ \text{definition of } \mathit{insert} \} \\
&\quad 1 : 2 : [3]
\end{aligned}$$

This took 9 evaluation steps.

The recurrence equation for the worst case time complexity of *sort xs* is given by $T_{isort}(n)$, where $n = \mathit{length}\ xs$, to be the following:

$$\begin{aligned} &T_{sort}\ 0 = 1 \\ &T_{sort}\ n = 1 + T_{insert}\ (n-1) + T_{sort}\ (n-1) \end{aligned}$$

This can be solved exactly by recognising that this is the equation for triangular numbers, the sum $1 + 2 + \ldots + n$ that Gauss famously solved as a child.

The story goes that Gauss solved this problem in 1786 when he was about nine years old, by coming up with this equation when his teacher set it as an exercise. The classic source of this was written for his memorial by von Waltershausen [1856].

$$\begin{aligned} T_{sort}\ n &= \tfrac{n \times (n+1)}{2} + 1 + n \\ &= \tfrac{(n+1) \times (n+2)}{2} \end{aligned}$$

As values of n grow, this complexity is dominated by the n^2 term in the equation, which is to say that in the worst case it has $O(n^2)$ behaviour.

These two examples have illustrated the key elements that will be repeated throughout this course: the statement of a problem, the design and implementation of an algorithm, and the analysis of time complexity. The remainder of these notes will focus on elaborating on these key elements and demonstrating key algorithmic principles that can be used to solve problems and analyse solutions.

2 Evaluation

> Perfection is achieved, not when there is nothing left to add, but when there is nothing left to take away.
>
> Antoine de Saint-Exupéry, 1939
> Terre des Hommes

A critical part of understanding algorithms is the ability to estimate the time it takes for an algorithm to provide a solution. Estimating this time requires a model of how much time it takes for instructions to be executed. This is known as the *cost model*. The exact cost model that it is used must of course depend on how instructions are executed in the first place. This might be the number of assignments that are made, the number of times values are compared with one another, the number of function calls that are made, and so on: the model can be adapted to suit particular needs. A very general cost model counts the number of *reductions* that are made, and that in turn requires an understanding of reduction strategies.

Given an expression there can are a number of different evaluation strategies that lead to a reduction. For instance, consider the evaluation of the following definition of a *minimum* function:

$$\begin{aligned}&\mathit{minimum} :: [\mathit{Int}] \to \mathit{Int}\\&\mathit{minimum} = \mathit{head} \circ \mathit{sortInt}\end{aligned}$$

How many steps are required to evaluate $\mathit{minimum}\ [3,1,2]$? The previous count showed that $\mathit{sort}\ [1,2,3]$ takes 9 steps, so one answer is that an additional step is required for the *head* function, thus 10 steps. However, there is a potential shortcut: after only 7 steps the value 1 is at the head of the list: so perhaps it could be extracted at that point instead. In short, the exact order of evaluation of certain expressions will change the number of steps required.

This chapter introduces a simple expression language and the main evaluation strategies that can be employed. Then, a means of carefully counting the number of reductions in a strict setting is discussed.

2.0.1 Expressions

The strategy to giving an overall cost to an algorithm is to give a series of rules that will assign a cost to a function by allocating a cost to its constituent parts. While Haskell is a language with many different syntactic features, at its core it can be simplified into a much smaller language.

To a first approximation, expressions, e in a simple programming language can be broken down into terms given by the following grammar:

$$\begin{array}{lll} e ::= & x & \text{-- variables} \\ & \mid\ k & \text{-- constants} \\ & \mid\ f\ e_1 \ldots e_n & \text{-- applications} \\ & \mid\ \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \text{-- conditionals} \end{array}$$

A function f in this language is assume to be defined by the form $f\ \ x_1 \ldots x_n\ =\ e$. Infix operations are written as $x + y$ instead of $(+)\ x\ y$. Primitive constants such as *True*, *False*, 0, 1, 2, are available, as are stand operations on them such as $\neg$, $\leqslant$, $(+)$, and $(\times)$. List constants and operations are also primitive, such as $[\,]$, $(:)$, *null*, *head* and *tail*.

While this does not cover all the syntactic constructs available in a fully featured language like Haskell, this covers a wide range of programs that can be broken down into these components.

2.0.2 Evaluation Order

There are two main forms of evaluation: *applicative evaluation* and *normal evaluation*. As stated before, different evaluation strategies may take different times to compute an expression.

- *Applicative order* works on the leftmost innermost reducible expression when evaluating. This amounts to evaluating arguments to a function before evaluating the function itself.
- *Normal order* works on the leftmost outermost reducible expression when evaluating. This amounts to evaluating the function before evaluating its arguments.

If they terminate, both of these strategies produce values in normal form. If a normal form for an expression exists, then normal order will reduce to that normal form. Applicative order, on the other hand, may not find the normal form since it may not terminate, but if it does terminate then the result will agree with normal order.

These properties of reduction strategies are given by the Church-Rosser Theorem, which is not explored further in this course. The classic result is by Church and Rosser [1936].

In a *strict* setting, evaluation occurs using applicative order. In a *lazy* setting, evaluation occurs using normal order. Furthermore, in functional languages no evaluation occurs in the body of a lambda abstraction, so all values that are produced are only weakly normalised.

To keep the analysis of algorithms simple, the standard assumption is that strict evaluation is used on arguments that are already in normal form. At times this assumption may be changed or questioned, but it serves as a good baseline for most of the algorithms that will be encountered in this course.

Even in an imperative language like C with eager evaluation strategies, not all statements are evaluated. Conditional expressions such as **if** statements and short-circuit operators only evaluate the necessary branches.

2.1 *Counting Carefully*

A proper account of how long it takes for a function to run is achieved by imposing an appropriate cost model. In a strict setting, it is possible to count the number of steps accurately. A simple model is to count the total number of times a non-primitive function is reduced.

To do this, the cost of evaluating a function f which takes n arguments will be given by the function $T(f)$ which also takes n arguments, such that $T(f)\ \ x_1 \dots x_n$ is the time it takes to evaluate $f\ x_1 \dots x_n$.

Now consider a given non-primitive function f that takes arguments $a_1, ..., a_n$ to produce an expression e. In other words, a definition of the form:

$$f\ a_1\ a_2 \dots a_n = e$$

If the arguments $a_1, ..., a_n$ are already evaluated, then the time time it takes to compute this function is $T(f)$, which can be expressed by:

$$T(f)\ a_1\ a_2 \dots a_n = T(e) + 1$$

This means that the time it takes to compute the application of the function f to its arguments is the time it takes to evaluate the body e, plus some constant measure of time.

Now the goal is to define the function $T(e)$ for the different expressions e.

Primitive A primitive function f that takes n arguments costs:

$$T(f)\ x_1 \dots x_n = 0$$

Variable A variable x costs:

$$T(x) = 0$$

Application An application $f\ e_1 \dots e_n$ costs:

$$T(f\ e_1 \dots e_n) = T(f)\ e_1 \dots e_n + T(e_1) + \dots + T(e_n)$$

Conditional A conditional expression costs:

$$T(\ \textbf{if}\ p\ \textbf{then}\ e_1\ \textbf{else}\ e_2) = T(p) + \textbf{if}\ p\ \textbf{then}\ T(e_1)\ \textbf{else}\ T(e_2)$$

With all these costs defined, applying the cost model thus proceeds in two phases. First, expressions are translated from ordinary Haskell to a smaller fragment that is easier to analyse, and once this is done the costs are calculated.

While this method of calculating costs is certainly possible, it is not practical in the sense that the process is too involved for even the simplest of algorithms. It does serve, however, to aid intuition when trying to understand the cost of a computation.

This was an involved process even for the simple case of strict evaluation. Wadler [1988] discusses an initial model for lazy evaluation, which involves first performing strictness analysis on a program. Complexity analysis in lazy functional languages was investigated more thoroughly by Sands [1989, 1990], which resulted in a calculi for the time analysis of functional programs.

3
Asymptotics

> All we have to decide is what to do with the time that is given us.
>
> J. R. R. Tolkien, 1954
> The Fellowship of the Ring

Giving an exact measure of how many steps it will take for a computation to terminate is not always an easy or even a necessary task. The exact cost of an arbitrary function could involve a recurrence equation that is every bit as intricate as the function itself. Instead, time complexity is usually given by placing a bound in terms of a simpler class of functions that approximate the actual running cost.

Hardy [1910, p24] introduced L-functions as a short name for the *logarithmico-exponential functions*. In earlier work [Hardy, 1905, p3] he used the term *elementary functions* but the term now loosely encompasses a larger family of functions that also includes trigonometric functions and their inverses. The work on elementary functions began with the treatment by Liouville [1833] of *algebraic functions* which are similar but do not include functions made up of logarithms or exponentials.

Arbitrary functions can be difficult to compare and analyse, yet for a large class of functions, the *logarithmico-exponential* functions, or *L-functions*, a useful complexity hierarchy can be constructed.

Definition 3.0.1 (L-function)**.** A *logarithmico-exponential function*, or *L-function* is a real, positive, monotonic, one-valued function on a real variable defined for all values greater than some definite value by a finite combination of algebraic symbols, exponentials, and logarithms, operating on real constants and the variable.

For instance, given the number n as input, the functions that return n, n^2, or $n\log(n)$ are all L-functions, whereas those that return $-n$, and $-(n^2)$ are not. Figure 3.1 plots the growth of various simple L-functions.

It's a picky point, but $f(n)$ is not a function, it is a value which is the result of applying the function f to n. That said, giving the value $f(n)$ for each n does describe the function f.

L-Functions are particularly useful because the values they produce are well-behaved as the input grows large. This is captured by the following theorem.

Theorem 3.0.1. *Any L-function f is ultimately continuous, of constant sign, monotonic, and as $n \to \infty$, the value $f(n)$ tends to one of 0, ∞, or some other definite limit.*

This theorem is useful in that it allows the relationship between L-functions to be categorised.

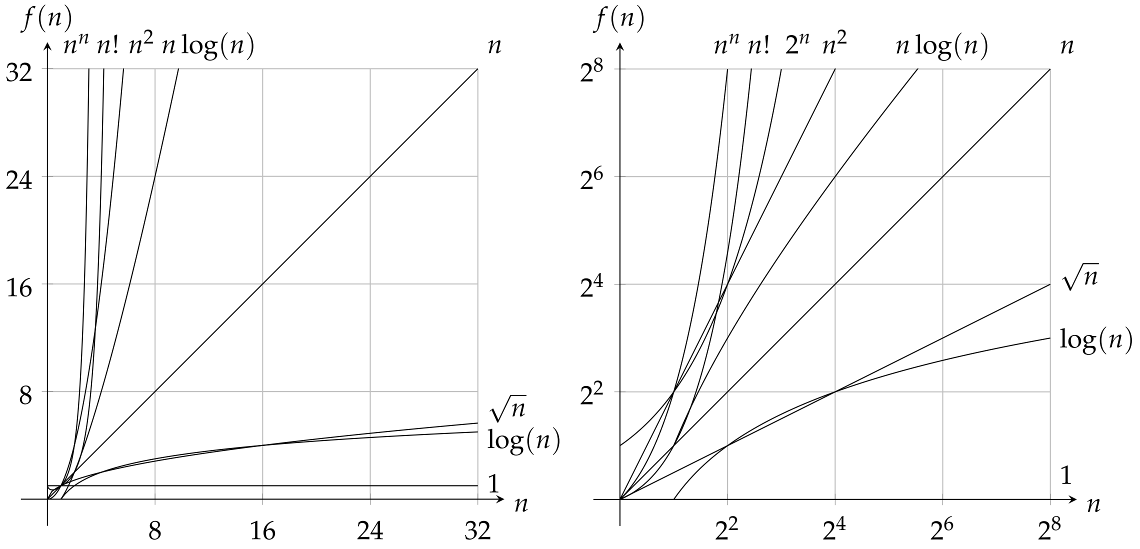


Figure 3.1: Comparison of various functions of n on linear and log scales.

3.1 *Du Bois-Reymond Notation*

The notation $f \prec g$ is due to du Bois-Reymond [1870]. This was later extended by Hardy [1910] to include the other relations presented here.

The first important concept when looking at complexity is the notion of the *rate of increase* of a function relative to another: a function can only be seen to grow quickly or slowly with respect to some other function. As is standard, the rate of increase of two functions can be understood as the ratio between them.

Now suppose f and g are L-functions, and consider the ratio of the L-function $f/g(n) = f(n)/g(n)$ as n tends to infinity. This gives rise to a family of operations, $\prec$, $\preccurlyeq$, $\asymp$, $\succcurlyeq$, and $\succ$ that can be used to compare functions.

There are many variations of these definitions depending on whether or not f is considered to be well-behaved. The assumption that f and g are L-functions is key to these particular definitions.

$$f \prec g \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{3.1}$$

$$f \preccurlyeq g \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \tag{3.2}$$

$$f \asymp g \iff 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \tag{3.3}$$

$$f \succcurlyeq g \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \tag{3.4}$$

$$f \succ g \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \tag{3.5}$$

The result of comparing two L-functions f and g is depicted in Figure 3.2, which shows the interval line of the real numbers extended with the upper bound ∞.

For L-functions f and g, Theorem 3.0.1 says that one of $f \prec g$, $f \asymp g$, or $f \succ g$ must hold, forming a trichotomy. There are a number of other properties which state that these operations behave much like inequalities.

First, $\prec$ and $\succ$ are converse:

$$f \prec g \iff g \succ f \tag{3.6}$$

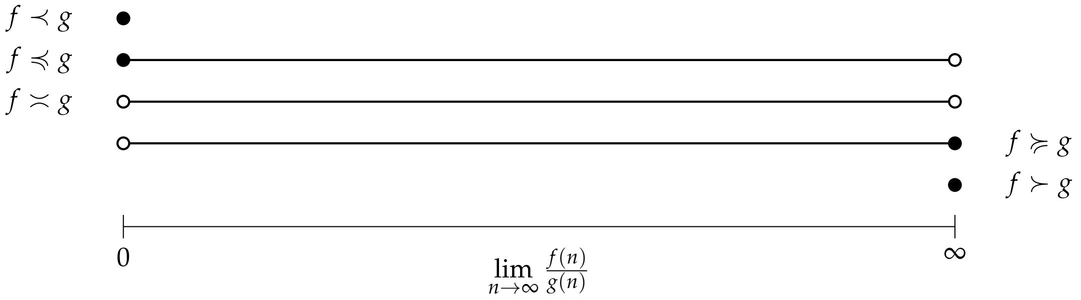


Figure 3.2: Interval lines showing the relationship between f and g.

In addition, the operations are transitive:

$$f \prec g \wedge g \prec h \implies f \prec h \tag{3.7}$$

$$f \preccurlyeq g \wedge g \preccurlyeq h \implies f \preccurlyeq h \tag{3.8}$$

There are many other properties that hold that are not listed here.

The following relations hold on functions of n, forming an (incomplete) hierarchy of L-functions:

$$1 \prec \log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec n^3 \prec n! \prec n^n$$

3.2 *Bachman-Landau Notation*

An alternative way of understanding the complexity of a function is to use Bachman-Landau notation, sometimes also called Big-O notation. Given a function f, its rate of growth can be measured by a family of functions related to some function g. This gives rise to the definitions of o, O, Θ, Ω, and ω which are operations which, given a function of n produce a set of functions.

These operations can be defined in terms of the du Bois-Reymond inequalities.

$$f \in o(g(n)) \iff f \prec g \tag{3.9}$$

$$f \in O(g(n)) \iff f \preccurlyeq g \tag{3.10}$$

$$f \in \Theta(g(n)) \iff f \asymp g \tag{3.11}$$

$$f \in \Omega(g(n)) \iff f \succcurlyeq g \tag{3.12}$$

$$f \in \omega(g(n)) \iff f \succ g \tag{3.13}$$

These sets can also be defined directly:

$$o(g(n)) = \{\ f\ |\forall\delta > 0.\exists n_0 > 0.\forall n > n_0.f(n) < \delta g(n)\ \} \tag{3.14}$$

$$O(g(n)) = \{\ f\ |\exists\delta > 0.\exists n_0 > 0.\forall n > n_0.f(n) \leqslant \delta g(n)\ \} \tag{3.15}$$

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) \tag{3.16}$$

$$\Omega(g(n)) = \{\ f\ |\exists\delta > 0.\exists n_0 > 0.\forall n > n_0.f(n) \geqslant \delta g(n)\ \} \tag{3.17}$$

$$\omega(g(n)) = \{\ f\ |\forall\delta > 0.\exists n_0 > 0.\forall n > n_0.f(n) > \delta g(n)\ \} \tag{3.18}$$

$$\tag{3.19}$$

Bachmann [1894, p. 401] introduced Big-O notation saying *"[...] wenn wir durch das Zeichen $O(n)$ eine Grösse ausdrücken, deren Ordnung in Bezug auf n die Ordnung von n nicht überschreitet; ob sie virklich Glieder von der Ordnung n in sich enthält, bleibt bei dem bisherigen Schlussverfahren dahingestellt."*

Landau [1909, p. 61] added the mnemonic little-o notation, stating *"O soll an Ordnung, o an „von kleinerer Ordnung" erinnern"*.

The definition of Ω is a slightly controversial topic. The original definition is due to Hardy and Littlewood [1914] and is slightly different to the version presented in this course. Later, Knuth [1976] decided to change the definition presented here. He says "Although I have changed Hardy and Littlewood's definition of Ω, I feel justified in doing so because their definition is by no means in wide use, and because there are other ways to say what they want to say in the comparatively rare cases when their definition applies." Later, Vitányi and Meertens [1985] pointed out that Hardy's definition has better complementary mathematical properties.

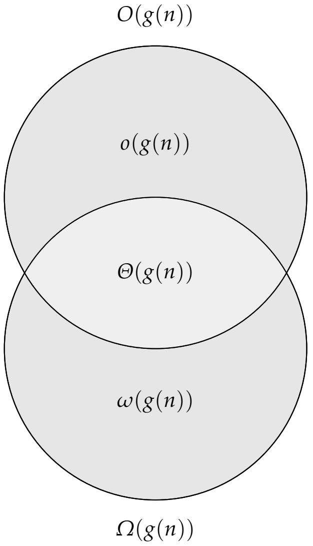


Figure 3.3: Venn diagram of asymptotic notation for the function g. The upper circle is $O(g(n))$, and the lower circle is $\Omega(g(n))$.

4
Lists

> There are two ways of constructing a software design: one way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies. The first method is far more difficult.
>
> C. A. R. Hoare, 1980

One prevalent model of computation is as a series of steps executed in sequential order. Since lists embody the notion of sequentiality as data, it should be little surprise that they are a fundamental datastructure. This chapter studies lists and their representations as an exercise in understanding the algorithms that accompany a datastructure.

In Haskell, lists receive special treatment, where their notation and definition are built into the language. That said, a definition of lists in Haskell would look like the following code.

```
data [a] = []
         | (:) a [a]
```

The **data** keyword indicates that a new datatype is being introduced. The type itself is called $[a]$, which can be constructed by means of the constructors, $[\,]$ for empty lists, and $(:)$ for adding an element to a list, introduced to the right of the equality symbol. The types of these constructors is:

```
[] :: [a]
(:) :: a → [a] → [a]
```

Given fully evaluated arguments, constructors are assumed to take constant time to construct values.

Unlike a homogeneous list that contains elements all of the same type, a heterogeneous list such as $[1, \mathit{True}, \texttt{'a'}, 5]$ contains values with a mixture of types. These cannot be defined using these constructors in Haskell.

Any homogeneous list can be expressed using these constructors. For instance the term $1 : 3 : 3 : 7 : [\,]$ is a list containing 4 numbers. The language provides syntactic sugar so that this list can also be written simply as $[1, 3, 3, 7]$, which is a welcome convenience. Dually, any list can be decomposed into these two constructors, which forms the basis of a simple pattern for designing operations on lists.

While the constructors of a list execute in constant time, it is easy to come up with operations that produce lists that take longer. For instance, consider how long it takes for the function $(+\!\!+)$ that appends two lists together:

Since Haskell is a lazy language, its lists are equivalent to the *streams* introduced by Landin [1965] in that they need not be finite datastructures. To avoid confusion, *list* will refer to finite datastructures in these notes.

$$\begin{aligned}
&(+\!\!+) :: [a] \to [a] \to [a] \\
&[\,] \qquad\;\; +\!\!+\, ys = ys \\
&(x : xs) +\!\!+\, ys = x : (xs +\!\!+\, ys)
\end{aligned}$$

The whole list must be traversed, so given $length\ \ xs\ =\ n$, then $T_{(+\!\!+)}(n) \in O(n)$.

It is worth noticing that the list ys is persisted in this datastructure: no values were harmed in the creation of the list $xs +\!\!+ ys$. This has the benefit that sharing ys across other computations costs nothing, but at the cost of having to reallocate the space for xs.

4.1 List Origami

The most common way of working with a list is to decompose it in a structured way, where the pattern of recursion follows the structure of the list itself. In the $(+\!\!+)$ example, the first parameter was broken down using case analysis into the two possible constructors. In the recursive case, the recursion was performed on the sublist in the constructor.

This pattern of recursion crops up frequently, and is captured by the *foldr* function:

$$\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b \\
&foldr\,f\ k\ [\,] \qquad\;\; = k \\
&foldr\,f\ k\ (x : xs) = f\ x\ (foldr\,f\ k\ xs)
\end{aligned}$$

This pattern of recursion is useful because it is the fundamental eliminator for lists. To understand its behaviour, it helps to see how it behaves for a given list:

$$foldr\,f\ k\ [x_1, x_2, ..., x_n] = f\ x_1\ (f\ x_2\ (...(f\ x_n\ k)))$$

When *foldr* is used with is a binary operator $(\diamond)$, the following holds:

$$foldr\ (\diamond)\ \epsilon\ [x_1, x_2, ..., x_n] = x_1 \diamond (x_2 \diamond (... \diamond (x_n \diamond \epsilon)))$$

Furthermore, when $(\diamond)$ is associative and ϵ is a neutral element, this is simply:

$$foldr\ (\diamond)\ \epsilon\ [x_1, x_2, ..., x_n] = x_1 \diamond x_2 \diamond ... \diamond x_n$$

One way to interpret this is that applying *foldr* is a destructor of lists, dual to the operations $(:)$ and $[\,]$ which are constructors. This can be seen by considering the effect of $foldr\ (:)\ [\,]$:

$$\begin{aligned} \mathit{foldr}\ (:)\ [\,]\ [x_1, x_2, ..., x_n] &= x_1 : x_2 : ... : x_n : [\,] \\ &= [x_1, x_2, ..., x_n] \end{aligned}$$

This does nothing to the list, so $\mathit{foldr}\ (:)\ [\,] = \mathit{id}$. In other words, destroying a list with its constructors leaves it unchanged.

As an example, the function *concat* takes a list of lists and appends their contents together maintaining their order in order to produce a single list. It is defined as a recursive function is as follows:

$$\begin{aligned} &\mathit{concat} :: [[a]] \to [a] \\ &\mathit{concat}\ [\,] \qquad\quad = [\,] \\ &\mathit{concat}\ (xs : xss) = xs + \!\!+\ \mathit{concat}\ xss \end{aligned}$$

This can be defined in terms of a *foldr*. To see how, it helps to see how the result is formed for some input:

$$\mathit{concat}\ [xs_1, xs_2, ..., xs_n] = xs_1 +\!\!+ (xs_2 +\!\!+ (... +\!\!+ (xs_n +\!\!+ [\,])))$$

This is precisely the shape of a *foldr*, so an alternative definition of *concat* is as follows:

$$\begin{aligned} &\mathit{concatr} :: [[a]] \to [a] \\ &\mathit{concatr} = \mathit{foldr}\ (+\!\!+)\ [\,] \end{aligned}$$

Many other functions can be defined as a *foldr* using this approach.

An obvious question at this point is to determine the complexity of *concat*. Assuming that $\mathit{length}\ xs_i = m$ for each i helps with the analysis, because each $(+\!\!+)$ operation deals with a list of the same length, and $T_{(+\!\!+)}(m) \in O(m)$.

$$\underbrace{xs_1}_{m} +\!\!+ (\underbrace{xs_2}_{m} +\!\!+ ... +\!\!+ (\underbrace{xs_n}_{m} +\!\!+ [\,]))$$

Annotating each list with its length helps to visualise that the overall time taken is $T_{concatr}(m, n) \in O(nm)$, since the time it takes to execute each $(+\!\!+)$ is dependent only on the size of its left argument.

4.2 Associativity Matters

Since $(+\!\!+)$ is associative with a neutral element $[\,]$, it is reasonable to ask if there could have been an alternative definition where the association is to the left:

A *monoid* is a set X that is equipped with an associative binary operation $(\diamond) : X \times X \to X$ and a neutral element $\epsilon : X$.

$$\mathit{concat}\ [xs_1, xs_2, ..., xs_n] = ((([\,] +\!\!+ xs_1) +\!\!+ xs_2) +\!\!+ ...) +\!\!+ xs_n$$

Just as *foldr* can be viewed as capturing the pattern of right-associated definitions, *foldl* can be viewed as capturing the pattern of a left-associated definitions.

The type of *foldl* takes a binary operation where the accumulation of the function is in the leftmost parameter.

$$\mathit{foldl}\ (\diamond)\ \epsilon\ [x_1, x_2, ..., x_n] = (((\epsilon \diamond x_1) \diamond x_2) \diamond ...) \diamond x_n$$

This can be expressed by the following definition:

$$\begin{aligned}
&\mathit{foldl} :: (b \to a \to b) \to b \to [a] \to b \\
&\mathit{foldl}\ f\ k\ [\,] \qquad\ \ = k \\
&\mathit{foldl}\ f\ k\ (x : xs) = \mathit{foldl}\ f\ (f\ k\ x)\ xs
\end{aligned}$$

The base case is the same, where the parameter k is the result. In the recursive case, *foldl* is tail recursive: it immediately calls itself where the accumulating parameter stores the result of applying the function f.

Using *fold* an alternative definition of *concat* is can be defined:

$$\begin{aligned}
&\mathit{concatl} :: [[a]] \to [a] \\
&\mathit{concatl} = \mathit{foldl}\ (+\!\!+)\ [\,]
\end{aligned}$$

It is possible to show by equational reasoning that *concatl* = *concatr*. Since the functions are equal, it may be tempting to assume that they also have the same time complexity. Alas, this is not the case.

To understand the complexity of *concatl* it helps to look at the size of the lists which are being appended to the left of each $(+\!\!+)$:

$$\underbrace{(\underbrace{(\underbrace{(\underbrace{[\,]}_{0} +\!\!+ xs_1)}_{m} +\!\!+ xs_2)}_{2m} +\!\!+ \ldots)}_{(n-1)m} +\!\!+ xs_n$$

This time the overall complexity is $T_{concatl}(m, n) \in O(n^2 m)$. The lesson here is that although *concatl* and *concatr* are *extensionally* equal, that is, the values they produce are the same, they are clearly *intensionally* different, that is, they produce these values in different ways.

Naively it would seem that this problem with associativity is resolved by simply taking *concatr* as the appropriate definition, since it has better characteristics. However, this does not avoid the fundamental issue: lists that are composed through a series of left-associated appends will suffer from bad performance. The trouble is that this happens very naturally a string that logs information is constructed by adding values to the end, and this ultimately leads to a left-associated list. Resolving this problem will require a change of representation of lists, which will be discussed in the next chapter.

5
Abstract Datatypes

> We must beware of needless innovation,
> especially when guided by logic.
>
> Winston Churchill, 1942

Lists can be defined in terms of a concrete instance where the specific constructors are listed and form the basis for pattern matching functions that operate on the structure. Although this demonstrates how a list is intended to behave, the problem is that certain operations on lists can end up with unavoidable inefficiencies in that representation. This chapter discusses how an abstract interface for lists can be created that can be instantiated to different concrete implementations with varying complexity characteristics.

```
class List list where
  fromList :: [a] → list a
  toList :: list a → [a]
  normalize :: list a → list a

  empty :: list a
  single :: a → list a

  cons :: a → list a → list a
  snoc :: list a → a → list a
  head :: list a → a
  tail :: list a → list a
  init :: list a → list a
  last :: list a → a

  isEmpty :: list a → Bool
  isSingle :: list a → Bool

  length :: list a → Int
  (++) :: list a → list a → list a

  (!!) :: list a → Int → a
```

Figure 5.1: The abstract interface of lists, given by the *List* type class

5.1 List Interface

An abstract definition of a list specifies a series of operations that work with lists. Key operations include *empty*, *cons*, *snoc*, *length*, (++), *head*, *tail*, *init*, *last*, (!!), though this is not exhaustive. The operations are specified by describing their effect on an underlying abstraction. In this case the underlying abstraction is the standard list type, since the goal is to provide alternative concrete implementations that behave similarly with respect to these operations.

Technically, not all of these operations are required to fully specify list behaviour: many of them can be defined in terms of the others. However, for some of these operations, an efficient implementation for certain list representations can only be given by directly manipulating the concrete implementation.

Haskell has fixed definitions for most of these functions in the *Prelude* already, so redefining them will cause ambiguity. To generalise these definitions into more abstract versions, the *Prelude* versions must be hidden:

```
import Prelude hiding
  (head, tail, init, last, (!!), length, (++))
```

Such hidden functions can still be used with a fully qualified name such as *Prelude.length* by importing *Prelude* qualified:

```
import qualified Prelude
```

The abstract list representation must contain these operations, and so they are given by a new typeclass called *List*, given fully in Figure 5.1. A type class outlines the signatures of functions that must be implemented for a valid instance. In this case, the *List* class takes a type parameter *list* which can be instantiated to a concrete list implementation type.

The key operations of this class are *toList* and *fromList*, which dictate the equivalence between the abstract list specification, which are standard lists of type $[a]$, and the concrete list representation, which are of values of type *list a*.

```
class List list where
  toList :: list a → [a]
  fromList :: [a] → list a
  ...
```

These are properly defined when the following holds:

$$toList \circ fromList = id$$

This establishes that an abstract list should not be modified when it is passed through its representation.

More generally, *toList* is an example of an *abstraction* function: it takes the concrete implementation to its abstract representation. Here, the type *list a* will be the concrete implementation, and $[a]$ is its abstract representation. Dually, the function *fromList* is a *representation* function since it determines the particular concrete representation of the abstract type.

It is tempting to give the specification as an isomorphism by stipulating that $fromList \circ toList = id$ also holds. However, this is too strong a requirement for certain representations where there may be multiple ways of representing a particular list. That said, $toList \circ fromList$ is a useful function for *normalizing* a representation into a canonical form. The fact can be captured by a law that also serves as the default implementation of *normalize*:

An isomorphism between two types a and b is given by a pair of functions $f :: a \to b$ and $g :: b \to a$ such that $f \circ g = id$ and $g \circ f = id$.

```
  ...
  normalize :: list a → list a
  normalize = fromList ∘ toList
  ...
```

The composition of these functions will normalize the list representation, although it is possible that this function could be specialised for a particular implementation.

Using *toList* and *fromList* also allows a simple specification of the other functions: the action of operations on the representation should be the same as their action on lists. For instance, here are some properties of functions that allow the construction of lists:

```
  ...
  empty :: list a
  empty = fromList [ ]
  cons :: a → list a → list a
  cons x xs = fromList (x : toList xs)
  single :: a → list a
  single x = fromList [x]
  snoc :: list a → a → list a
  snoc xs x = fromList (toList xs ++ [x])
  ...
```

For instance, the *tail* function is specified by the following property:

$$tail = toList \circ tail \circ fromList$$

This says that the result of the *tail* function on regular lists is the same as first converting from its representation, applying tail to that representation, and the converting back to a list.

The other operations are specified in a similar way, and left as an exercise for the reader.

5.2 Default Lists

The abstract representation of lists is given by $[a]$, the standard list type. Unsurprisingly, standard lists can also serve as a perfectly valid implementation. The functions *toList* ad *fromList* are simply identities:

```
instance List [ ] where
  fromList = id
  toList   = id
```

The normalization function is also an identity:

```
  normalize = id
```

The instance for regular lists can either be given directly by reference to the standard function definitions in the *Prelude*, or by redefining them in place. For instance, here are two alternative implementations of the *length* function:

```
  length :: [a] → Int
  length [ ]      = 0
  length (x : xs) = 1 + length xs
```

By default Haskell will reject code with signatures in the instances. They are useful for humans and are enabled here by using the {-# LANGUAGE InstanceSigs #-} language extension.

This is the usual definition via pattern matching. The alternative is to simply call the version that is already defined in the *Prelude*:

```
  length :: [a] → Int
  length = Prelude.length
```

At this point, it is a worthwhile exercise to get a sense of the time complexities of the standard functions. In this case, given $length\ xs = n$, the time it takes to compute $length\ xs$ is given by $T_{length} \in O(n)$.

5.3 Tree Lists

A binary tree with values at its leaves can be considered to be a list, where an in-order traversal of the list from left to right corresponds to the order of the list elements. The benefit of this representation is that appending two trees together is achieved by simply placing them under a parent fork.

5.4 Difference Lists

Difference lists are a way of getting constant time *cons*, *snoc*, and $(+\!\!+)$. The trade-off is that now certain other operations become more expensive.

The main insight of a difference list is to replace the $(+\!\!+)$ operation with $(\circ)$, function composition, so that appending lists together always ends up in a right-associated list. Right association is ensured because of the definition of function composition:

The difference list representation was first introduced to functional programming by Hughes [1986], but the technique was actually predated quite some time before by Cayley [1854] who came up with the modern mathematical definition of groups.

$$\begin{aligned}
&(\circ) :: (b \to c) \to (a \to b) \to (a \to c)\\
&(g \circ f)\ x = g\ (f\ x)
\end{aligned}$$

The composition $g \circ f$ ensures that f is applied to the argument first, followed by g: application happens from right to left. If $f = (xs{+\!\!+})$, $g = (ys{+\!\!+})$, and $h = (zs{+\!\!+})$, then their composition applied to the empty list gives:

$$((zs{+\!\!+}) \circ (ys{+\!\!+}) \circ (xs{+\!\!+}))\ [\,] = zs +\!\!+ (ys +\!\!+ (xs +\!\!+ [\,]))$$

In other words, the composition of these list appending functions produces a nest of right-associated appends, which is desirable.

More abstractly, the goal is to represent of a list of type $[a]$ by a function of type $[a] \to [a]$. To this end a new type is created that defines *DList a* with a single constructor that contains the list representation:

$$\textbf{newtype}\ \mathit{DList}\ a = \mathit{DList}\ ([a] \to [a])$$

The idea is to design the representation function *fromList* and append operation $(+\!\!+)$ so that the following should hold:

$$\begin{aligned}
&\quad \mathit{toList}\ (\mathit{fromList}\ zs +\!\!+ \mathit{fromList}\ ys +\!\!+ \mathit{fromList}\ xs)\\
&= \quad \{\ \text{definition of}\ \mathit{fromList} :: a \to \mathit{DList}\ a\ \}\\
&\quad \mathit{toList}\ (\mathit{DList}\ (zs{+\!\!+}) +\!\!+ \mathit{DList}\ (ys{+\!\!+}) +\!\!+ \mathit{DList}\ (xs{+\!\!+}))\\
&= \quad \{\ \text{definition of}\ (+\!\!+) :: \mathit{DList}\ a \to \mathit{DList}\ a \to \mathit{DList}\ a\ \}\\
&\quad \mathit{toList}\ (\mathit{DList}\ ((zs{+\!\!+}) \circ (ys{+\!\!+}) \circ (xs{+\!\!+})))\\
&= \quad \{\ \text{definition of}\ \mathit{toList} :: \mathit{DList}\ a \to [a]\ \}\\
&\quad ((zs{+\!\!+}) \circ (ys{+\!\!+}) \circ (xs{+\!\!+}))\ [\,]\\
&= \quad \{\ \text{definition of}\ (\circ) :: ([a] \to [a]) \to ([a] \to [a]) \to ([a] \to [a])\ \}\\
&\quad zs +\!\!+ (ys +\!\!+ (xs +\!\!+ [\,]))
\end{aligned}$$

Note carefully that the append function $(+\!\!+)$ used here has type:

$$(+\!\!+) :: \mathit{List}\ \mathit{list} \Rightarrow \mathit{list}\ a \to \mathit{list}\ a \to \mathit{list}\ a$$

This has a constraint that states that *list* must be a member of the *List* class, which allows it to work on different types, as annotated in the calculation.

With this intuition in mind, the following definitions come out:

$$\begin{aligned}
&\textbf{instance}\ \mathit{List}\ \mathit{DList}\ \textbf{where}\\
&\quad \mathit{toList} :: \mathit{DList}\ a \to [a]
\end{aligned}$$

$$
\begin{aligned}
&\mathit{toList}\ (\mathit{DList}\ \mathit{fxs}) = \mathit{fxs}\ [\,] \\
&\mathit{fromList} :: [a] \to \mathit{DList}\ a \\
&\mathit{fromList}\ \mathit{xs} = \mathit{DList}\ (\mathit{xs}{+\!\!+}) \\
&({+\!\!+}) :: \mathit{DList}\ a \to \mathit{DList}\ a \to \mathit{DList}\ a \\
&\mathit{DList}\ \mathit{fxs} \mathbin{+\!\!+} \mathit{DList}\ \mathit{fys} = \mathit{DList}\ (\mathit{fxs} \circ \mathit{fys})
\end{aligned}
$$

The intention is that variables such as *fxs* represent functions on lists, so that $\mathit{fxs} = (\mathit{xs}{+\!\!+})$.

6
Divide & Conquer

> Divide et impera.
>
> Philip II of Macedonia, 382–336 BC

Divide and conquer is one of the most fundamental and useful algorithmic strategies. It consists of three parts:

1. Divide a problem into subproblems.

2. Divide and conquer subproblems into subsolutions.

3. Conquer subsolutions into a solution.

This is a recursive definition where it is assumed that certain problems are small enough to be solved without dividing them further.

6.1 Merge sort

A classic example of divide and conquer is merge sort:

```
msort :: [Int] → [Int]
msort [ ]  = [ ]
msort [x] = [x]
msort xs   = merge (msort us) (msort vs)
   where (us, vs) = splitAt (n ‘div‘ 2) xs
            n = length xs
```

The *splitAt* function is used to divide the problem into subproblems, and the *merge* solution is the conquer step that merges two sorted lists:

```
merge :: [Int] → [Int] → [Int]
merge [ ] ys    = ys
merge xs [ ]    = xs
merge (x : xs) (y : ys)
   | x ⩽ y       = x : merge xs (y : ys)
   | otherwise = y : merge (x : xs) ys
```

Two of the cases of *msort* are base cases: when the list is empty or singleton. This is critical to the termination of a divide and conquer strategy: the divide step must make the solution smaller, and the algorithm must be able to directly solve small enough problems.

The complexity of *msort* can be calculated as follows:

$$\begin{aligned}
T_{msort}(0) &= 1\\
T_{msort}(1) &= 1\\
T_{msort}(n) &= T_{length}(n) + T_{splitAt}(\frac{n}{2}) + T_{merge}(\frac{n}{2}) + 2 \times T_{msort}(\frac{n}{2})
\end{aligned}$$

Solving this recurrence gives $T_{msort}(n) \in \Theta(n \log n)$.

6.2 Quicksort

The *quicksort* algorithm is another example of a divide and conquer algorithm that also sorts values:

```
qsort :: [Int] → [Int]
qsort []       = []
qsort [x]      = [x]
qsort (x : xs) = qsort us ++ [x] ++ qsort vs
  where (us, vs) = partition (<x) xs
```

The divide step is given by the *partition* function, which inspects each element in a list with a given predicate, in order to partition the list into all elements that satisfy the predicate, and all elements that do not.

One question is whether to use $<x$ or $\leqslant x$ in the partition. Since the head of the list is used as the pivot, $<x$ is desirable so that this is a *stable* sorting.

```
partition :: (a → Bool) → [a] → ([a], [a])
partition p xs = (filter p xs, filter (¬ ∘ p) xs)
```

The conquer step is more subtle to spot since it simply places the *pivot* element x between the two partitions.

Choosing a good pivot turns out to be essential for a good quicksort.

Although the best case performance of *qsort* is the same as *msort*, it can do very badly if the pivot is not well chosen. In the worst case, the analysis is:

$$\begin{aligned}
T_{qsort}(n) &= T_{partition}(n-1) + T_{++}(n-1) + T_{qsort}(n-1) + T_{qsort}(0)\\
&= c \times n + T_{qsort}(n-1)\\
&= c \times n + c \times (n-1) + \ldots + c \times 1\\
&= c \times \frac{n \times (n+1)}{2}\\
&= c \times \frac{n^2 + n}{2}
\end{aligned}$$

In other words, $T_{qsort}(n) \in O(n^2)$ in the worst case.

7
Dynamic Programming

> A man who dares waste one hour of time
> has not discovered the value of life.
>
> Charles Darwin, 1836

Dynamic programming was first introduced by Bellman [1957] as a technique to efficiently calculate the exact solutions to certain recursive problems. The slogan for dynamic programming algorithms is to *trade space for speed*: the table takes space in memory to construct, but results in a much faster algorithm.

"Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities" — Richard Bellman (with other good excerpts in Dreyfus [2002]).

A key ingredient of dynamic programming is *memoization*. This technique, coined by Michie [1968], is a means of storing the result of a function call so that it can be used again later. Dynamic programming works by optimising the runtime performance of a recursive algorithm that has overlapping subproblems. The speedup comes from storing the subsolutions through memoization for later use instead of recomputing them every time they are needed.

The strategy is developed in two stages:

1. Write an inefficient recursive algorithm that solves the problem.
2. Improve efficiency by storing intermediate shared results.

As a simple example, dynamic programming can be applied to speed up a naive implementation of the Fibonacci function.

Each Fibonacci number for a given value n is given by $\mathit{fib}\ n$ given by the following recursive algorithm:[1]

$$\begin{aligned}
&\mathit{fib} :: \mathit{Int} \rightarrow \mathit{Integer}\\
&\mathit{fib}\ 0 = 0\\
&\mathit{fib}\ 1 = 1\\
&\mathit{fib}\ n = \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)
\end{aligned}$$

This code is remarkably inefficient, since there are repeated calls to computations that recalculate the same value. Figure 7 shows that $\mathit{fib}\ 8$ is called twice in the calculation of $\mathit{fib}\ 10$. In turn, $\mathit{fib}\ 7$ is called three times, $\mathit{fib}\ 6$ five times, and so on.

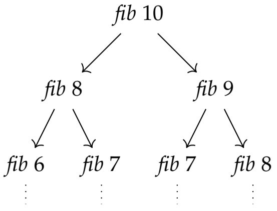


Figure 7.1: Call tree of *fib* 10, showing multiple repeated computations.

As an approximation, assume that the two subcalls of *fib* have the same cost, giving the recurrence relation:

$$T_{\mathit{fib}}(n) \leqslant 1 + 2 \times T_{\mathit{fib}}(n-1)$$

[1] Notice that this returns an *Integer* instead of *Int*. The function *fib* grows very quickly: at *fib* 93 there is already overflow on a 64 bit *Int*. The *Integer* gives access to values with unbounded precision.

Solving this recurrence gives $T_{fib}(n) \in O(2^n)$. This is remarkably expensive and due to the fact that there are large overlaps in the solutions of subproblems, which keep getting recalculated.

There are many different approaches to making this faster. The obvious one is to store additional information:

```
fib2 :: Int → Integer
fib2 n = go n 0 1 where
  go 0 x y = x
  go n x y = go (n − 1) y (x + y)
```

The catch is that each call to *fib*$_2$ must recalculate all previous results; asking for *fib* 10000 followed by *fib* 10001 will perform a lot of repeated computation.

The idea behind *dynamic programming* is to store common subsolutions to subproblems so that they can be reused in later computations.

Here is a simplified exposition of the idea:

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fib3 :: Int → Integer
fib3 n = fibs !! n
```

The function *fibs* produces a lookup table of all the values that have been computed so far. Asking for successive values of *fib* in a session will reuse previous results. The problem here, however, is that *fibs* is a list, which offers access to an element n in $O(n)$ time. An array would take only constant time to access precomputed elements.

In our setting an array can be created with the function *array*, that builds an array from a list containing indices and their values.

```
array :: Ix i ⇒ (i, i) → [(i, a)] → Array i a
```

The *Array* type comes equipped with an operation that can look up values in constant time:

```
(!) :: Ix i ⇒ Array i a → i → a
```

Given an array a and an index i, the result of $a \,!\, i$ is the value of the array a at index i. This returns in constant time, or fails catastrophically if the index is out of bounds.

A more traditional way of presenting a dynamic programming algorithm is to construct a table in an array containing the required results, and then to pull values out of the table as required. More concretely, the computation of *fib* n, involves defining an array containing the values of *fib* for *all* numbers from 0 up to n.

```
table :: Int → Array Int Integer
table n = array (0, n) [(0, 0)
                       ,(1, 1)
```

The *Array Int Integer* type represents an array indexed by *Int* containing values of type *Integer*.

The function (!) provides constant-time random access.

The function *array* takes a range (u, v) of values as its bounds, and a list of pairs where each pair (i, x) is used to place x at index i in the table.

```
              ,(2, table ! 0 + table ! 1)
              ,(3, table ! 1 + table ! 2)
              ,...]
```

Notice that every element of *table* refers to solutions of previous problems that eventually lead to a base case: there are no circular references so the self-referential definition is well-defined. Here is a version of *fib* that builds the right *table* and returns the last element of the array.

This is, of course, not the best way to calculate Fibonacci numbers (which can be done in sublinear time), but it illustrates how a recursive algorithm can be made more efficient.

```
fib′ :: Int → Integer
fib′ n = table ! n
  where
    table :: Array Int Integer
    table = tabulate (0, n) memo

    memo 0 = 0
    memo 1 = 1
    memo n = table ! (n − 1) + table ! (n − 2)
```

It is important that *memo* is in the same level of scope as *table*: if it were top-level then a new table would be created on each call!

The table given by *tabulate* (x,y) f contains the results of applying f to all the values between x and y. It is implemented as an array which gives constant time access to its elements.

The constraint *Ix i* allows the values of type i to be drawn from those given by the *range* function, as well as enabling values to be indexed over in an array. This allows arrays to be indexed by types other than *Int*. For instance, a valid index is a tuple (Int, Int) for a two-dimensional array indexed by pairs of *Int*s.

```
tabulate :: Ix i ⇒ (i, i) → (i → a) → Array i a
tabulate (u, v) f = array (u, v) [(i, f i) | i ← range (u, v)]
```

The cost of building this table is the sum of all the individual calls of f. The key to efficiency is that the function f can itself refer to the table that is being constructed. If the cost of f is constant, such that $T_f(i) \in O(1)$, and the table has n elements, then the cost of its construction is $T_{table}(n) \in O(n)$.

In the code above, *memo* is a local version of *fib* that finds values in the table rather than by recursion. The function takes constant time since $(!)$ is a constant time operation. Thus, the time complexity of evaluating *fib′* n is given by:

$$T_{fib'}(n) = 1 + T_{table}(n) + T_{!}(n)$$

where $T_{table}(n)$ is the time it takes to construct the table, and $T_{(!)}(n)$ is the time it take to look up a value in that table. Therefore, the overall cost is $T_{fib'}(n) \in O(n)$, which is much better than before.

```
import Dynamic (tabulate)
import Data.Array

fromList :: [a] → Array Int a
fromList xs = listArray (0, length xs − 1) xs
```

8
Edit-Distance

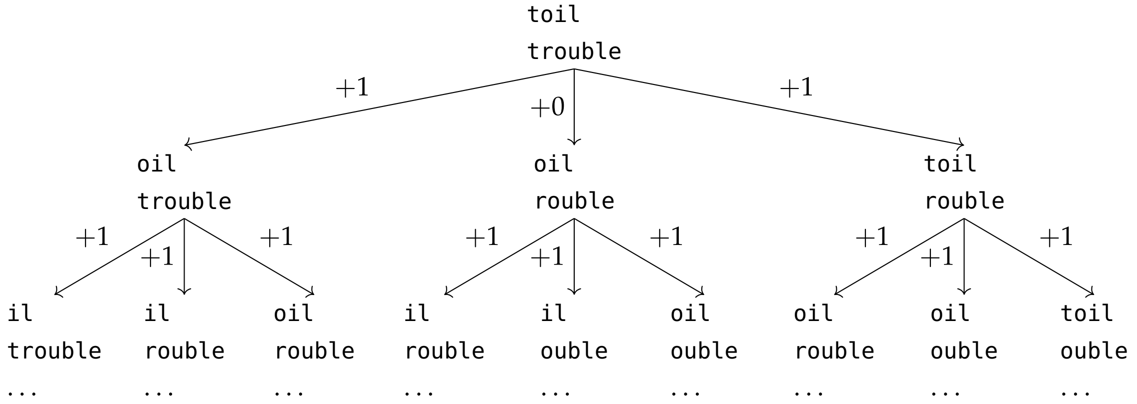
The *Edit-Distance Problem* concerns itself with finding the so-called *Levenshtein* distance between between two strings: the number of insertions, deletions, and update it takes to turn one string into another.

Notice that the problem can be simplified by considering only deletions and updates: an insertion of a character into one string to make a match is the same as deleting that character in the other string.

One way to visualise the edit distance problem is to draw a tree where each node contains the two strings that are currently being compared. Moving to the left child represents a deletion of the first character of the first string, and moving to the right child represents a deletion of the first character of the second string. Both of these moves add one to the edit distance cost.

Moving to the middle child represents matching the first characters of both strings: if those characters are the same then the cost is the same as that of the strings with those characters removed. Otherwise, an update is required to make those characters match, and the cost is the same as comparing the tails of the strings.

Figure 8.1: Edit distance tree



Finding the minimum edit distance cost is achieved by finding the lowest cost path starting at the root of the tree, and descending to a leaf.

Notice, however, that this tree involves quite a lot of repetition:

the strings obtained by deleting the first character of the first string and then second string, in that order, is the same as deleting both at the same time, which is the same right then left. This is what makes this a perfect candidate for dynamic programming.

Before going for an efficient version of the program, it is easier to write a recursive algorithm that solves this problem:

```
dist :: String → String → Int
dist xs [ ] = length xs
dist [ ] ys = length ys
dist xxs@(x : xs) yys@(y : ys) = minimum [dist xxs ys + 1
                                         ,dist xs yys + 1
                                         ,dist xs ys + if x ≡ y then 0 else 1]
```

The *minimum* function is called to consider the cheapest of the three choices at each node. Each choice is modelled by a recursive call to the strings with the appropriate character deleted, and has its corresponding cost added.

While this code gives the correct answer, it is terribly inefficient: there is a branching factor of 3, and the path from the root to the leaves is $|m + n|$, where m is the length of *xs* and n is the length of *ys*. This gives an overall complexity of $O(3^{|m+n|})$.

This complexity can be improved by applying dynamic programming so that subsolutions can be reused between invocations. Before going to a dynamic programming solution, however, a little work is needed. In *dist*, the recursive calls make use of different *String* parameters, and these make a very poor choice of index for an array.

To see why a *String* is a bad index, consider how the index might be constructed to contain all strings.

```
dist′ :: String → String → Int → Int → Int
dist′ xs ys i 0 = i
dist′ xs ys 0 j = j
dist′ xs ys i j  = minimum [dist′ xs ys i (j − 1) + 1
                           ,dist′ xs ys (i − 1) j + 1
                           ,dist′ xs ys (i − 1) (j − 1) + if x ≡ y then 0 else 1]
  where
    m = length xs
    n  = length ys
    x  = xs !! (m − i)
    y  = ys !! (n − j)
```

This code now uses appropriate values for indexing.

The transformation to an efficient dynamic problem is quite routine. The recursive calls are replaced by lookups into the *table*, and the table is populated with the *memo* function, which mirrors the recursive definition.

```
dist″ :: String → String → Int
dist″ xs ys = table ! (m, n)
  where
    table = tabulate ((0, 0), (m, n)) (uncurry memo)
```

```
memo :: Int → Int → Int
memo i 0 = i
memo 0 j = j
memo i j  = minimum [table ! (i, j − 1) + 1
                    , table ! (i − 1, j) + 1
                    , table ! (i − 1, j − 1) + if x ≡ y then 0 else 1]
  where
    x = axs ! (m − i)
    y = ays ! (n − j)

m = length xs
n = length ys

axs, ays :: Array Int Char
axs = fromList xs
ays = fromList ys
```

One small adjustment is that lookup of the values in table should be done in constant time. To achieve this, arrays *axs* and *ays* are created from *xs* and *ys*, and values *x* and *y* are drawn from those using an array lookup.

To estimate the running cost of this version, note first that there are $m \times n$ entries in the table. Each entry costs a constant time to create, so the overall cost is $O(mn)$.

9
Bitonic Travelling Salesman

> An investment in knowledge pays the best interest.
>
> Benjamin Franklin

The *travelling salesman* is a classic algorithmic problem. Given a set of cities connected by roads, the task is to plan the shortest route of a salesman along these roads who, starting in his home city, must visit each other city exactly once before returning home.

While this problem is difficult to solve efficiently, a variation called the *bitonic travelling salesman* problem has a nice dynamic programming solution. The bitonic travelling salesman has a similar goal: each city must be visited exactly once before returning home. The difference is that the cities are on a Euclidian plane, are all connected directly to one another, and the path that the salesman must take is a *bitonic tour*. To keep things simple, assume that there are $m + 1$ cities sorted from left to right, so that $p_0, \ldots, p_m$ determines their locations. The distance between two cities indexed by i and j can is also given in terms of $\delta\ i\ j$.

Since this is a Euclidian plane it must be the case that the two paths never cross: a shorter path between four points can always be found that has no crossovers, this fact isn't directly used in this implementation.

A bitonic tour from p_0 to p_n is a path that visits the points $p_0, \ldots, p_n$ exactly once with the property that it starts at the leftmost point p_0, heads only to the right before reaching the rightmost point p_n, and then heads only to the left before reaching back to the leftmost point.

The problem is solved with the function *bitonic* which, given the distance δ between any two points and an index n, calculates the shortest bitonic tour from p_0 to p_n. The solution to the bitonic travelling salesman problem is thus the result given by *bitonic* $\delta\ m$.

To construct this function, consider the properties of the shortest bitonic tour from p_0 to p_n. Each point is connected with two others and, apart from the endpoints, this must be with one point somewhere to the left, and with one point somewhere to the right.

Focusing on the endpoints, p_1 must be connected with p_0 since there are no other points to the left, and similarly p_{n-1} must be connected with p_n. Furthermore, consider the longest contiguous path $p_k, p_{k+1}, \ldots, p_{n-1}, p_n$, where $1 \leqslant k \leqslant n - 1$, that flows in one direction. Since it is an endpoint, p_n must be connected with the point p_{k-1} in the other direction.

The shortest bitonic tour must contain all of these edges, in addition to the edges in a path that connect p_0 with p_k in one direction and that connect p_{k-1} with p_0 in the other with no overlapping nodes except at p_0. Furthermore, the sum of the lengths of these paths must be the shortest. This is precisely given by the shortest bitonic tour from p_0 to p_k where the edge between p_k and p_{k-1} has been removed.

To summarize, if k is supplied then the length shortest bitonic tour can be calculated with the following equation:

```
bitonic δ n = bitonic δ k − δ (k − 1) k
                        + δ (k − 1) n
                        + sum [δ i (i + 1) | i ← [k . . n − 1]]
```

The trouble is that k is not given up front, so it must be found.

The desired value of k is bounded between 1 and $n - 1$. Additionally, it must be the one that minimises this equation. This directly gives rise to the following recursive definition of *bitonic*:

```
bitonic :: (Int → Int → Double) → Int → Double
bitonic δ 0 = 0
bitonic δ 1 = 2 × δ 0 1
bitonic δ n =
  minimum [ bitonic δ k − δ (k − 1) k
                         + δ (k − 1) n
                         + sum [δ i (i + 1) | i ← [k . . n − 1]]
          | k ← [1 . . n − 1]]
```

This code can be tested by creating some examples:

```
type Point = (Double, Double)
example :: Array Int Point
example = array (0, 3) (zip [0 . .]
  [(0, 0), (3, 4), (4, 3), (7, 7)])

dist :: Array Int Point
     → Int → Int → Double
dist ar i j = sqrt (x × x + y × y) where
  (xi, yi) = ar ! i
  (xj, yj) = ar ! j
  x = xi − xj
  y = yi − yj
```

To use this simply calculate the value *bitonic′* (*dist example*) 3, which is 20.

Looking at the recursive structure of this algorithm, it should be clear that it is ripe for dynamic programming, since there are many repeated calls to *bitonic* δ k for many different values of k.

The tabulated version of this function will allocate an array of $m + 1$ elements, one value for each point index. Filling each element in this array will need to search for k which will take linear time, assuming the summations can all be done in constant time, which brings the overall complexity algorithm to $O(m^2)$ time to compute.

9.0.1 Solution Extraction

Before working on the tabulated version it is worth considering how this recursion can be modified to give rise not only to the length of the shortest bitonic tour, but also to the order in which the points are visited. A simple way of achieving this is to additionally store a list of the paths, where the paths are represented by a pair of indices.

While the logic of the program remains the same, this change of type will require most of the code to be rewritten. An alternative is to apply a little abstraction, by looking at which operations the function *bitonic* requires: as well as applying addition and negation the minimum is needed, which in turn implies an ordering and a zero element. Overriding these operations can be achieved relatively easily by creating a new datatype that captures the required data, and implementing the relevant classes.

Technically, a valid *Num* instance also needs implementations of $(\times)$, *abs*, *signum* and a complete *fromInteger*, but these are not needed here and the convenience is simply too tempting.

```
data Path = Path Double [(Int, Int)]
  deriving (Show, Eq, Ord)

instance Num Path where
  Path d₁ ps₁ + Path d₂ ps₂ = Path (d₁ + d₂) (ps₁ ++ ps₂)
  Path d₁ ps₁ − Path d₂ ps₂ = Path (d₁ − d₂) (ps₁ \\ ps₂)

  fromInteger 0 = Path 0 [ ]
```

An invariance that will prove useful is to ensure that the components of pairs are in order, where (i, j) ensures that $i \leqslant j$, since this will make it easier to remove edges.

Now the new version of *bitonic′* is largely the same as before.

```
bitonic′ :: (Int → Int → Double) → Int → Path
bitonic′ δ 0 = Path 0 [(0, 0)]
bitonic′ δ 1 = Path (2 × δ 0 1) [(0, 1), (0, 1)]
bitonic′ δ n =
    minimum [ bitonic′ δ k − δ′ (k − 1) k
                           + δ′ (k − 1) n
                           + sum [δ′ i (i + 1) | i ← [k . . n − 1]]
            | k ← [1 . . n − 1]]
  where
    δ′ :: Int → Int → Path
    δ′ i j = Path (δ i j) [(min i j, max i j)]
```

The main difference is the use of δ', which uses δ to construct a path with the relevant distance and point information whilst normalizing the indices in pairs.

The tabulated version then follows the standard recipe: the recursive call is replaced with a lookup to a table, and the shared variable δ is lifted out.

```
bitonic″ :: (Int → Int → Double) → Int → Path
bitonic″ δ n = table ! n where
  table = tabulate (0, n) mbitonic

  mbitonic :: Int → Path
  mbitonic 0 = Path 0 [(0, 0)]
  mbitonic 1 = Path (2 × δ 0 1) [(0, 1), (0, 1)]
  mbitonic n =
      minimum [ table ! k − δ′ (k − 1) k
                          + δ′ (k − 1) n
                          + sum [δ′ i (i + 1) | i ← [k . . n − 1]]
              | k ← [1 . . n − 1]]
    where
      δ′ :: Int → Int → Path
      δ′ i j = Path (δ i j) [(min i j, max i j)]
```

Although this is clearly a better version than the recursive one, there are still some modifications needed for the complexity to be

$O(m^2)$. In particular, the cost of summing is not constant here since a list of values must be traversed. That said, calculating these results up-front and storing them for later retrieval is easily done quickly enough. Furthermore, the representation of paths is convenient but not optimal.

10
Amortized Analysis

No man is an Iland,
intire of itselfe;
every man is a peece of the Continent,
a part of the maine;

John Donne, 1624
Devotions upon Emergent Occasions

The complexity of the algorithms studied so far has been understood in terms of a single executions of the algorithm in isolation. Sometimes, however, the cost of an algorithm must be understood in its wider context. For instance, the application of an operations on a datastructure can interact with it in such a way that the cost of later operations is affected. This chapter explores *amortized analysis*, which takes gives the cost of an operation in the context of a sequence of previous operations on a datastructure.

10.1 Deques

In an ordinary list adding elements to the back is expensive: as shown in the implementation of *snoc*:

```
snoc :: [a] → a → [a]
snoc xs y = xs ++ [y]
```

The complexity of this, inherited from $(+\!\!+)$, is $O(n)$ where n is the length of *xs*. As a consequence elements should only be added to the end of a list sparingly.

Of course, if a list is grown only adding elements to the rear in this way, then it is preferable to add the elements to the front instead, and eventually reverse the result. However, this might not always be possible: perhaps elements are added to the front and back in alternation.

A double ended queue, or *deque*, sometimes also called a symmetric list, is a queue where elements can be added both at the front and at the back efficiently, and is in this sense double ended. The key insight is to represent the datastructure as a pair of queues: one starting from the front, and one from the back.

```
data Deque a = Deque [a] [a]
```

The idea is that the list *Deque xs sy* contains two components *xs* and *sy*, which together form the list with all the elements in *xs* followed by the reversed elements *sy*. In order to ensure that the representation remains balanced enough, there will occasionally have to be some shuffling of elements between the lists.

The goal will be to implement a *List Deque* class instance which fulfills the *List* class operations (Figure 5.1). The instance for *Deque* gives what is required:

```
instance List Deque where
  toList :: Deque a → [a]
  toList (Deque xs sy) = xs ++ reverse sy
```

With this in place, the next function to consider is *fromList*:

```
fromList :: [a] → Deque a
...
```

Now there are many different options. To guide the implementation, a deque datastructure *Deque xs sy* maintains the following invariance on *xs* and *sy*, later on it will be shown that it is essential for keeping the operations efficient as a whole:

$$isEmpty\ xs \Rightarrow isEmpty\ sy \lor isSingle\ sy$$
$$isEmpty\ sy \Rightarrow isEmpty\ xs \lor isSingle\ xs$$

These properties say that if one of the lists in the implementation is empty, then the other must contain at most one element. These properties are assumed and maintained by all the operations.

It is easy to construct a cheap *fromListNaive* function that creates a deque in constant time:

```
fromListNaive :: [a] → Deque a
fromListNaive xs = Deque xs []
```

However, this does not maintain the invariance and instead places the whole list into the first component.

A different version that respects the invariance is the following, which splits the list in two and puts half in each side:

```
fromList xs = Deque ys (reverse zs)
  where (ys, zs) = splitAt (length xs `div` 2) xs
```

This costs $O(n)$ when *xs* has length *n*, as would any version that splits the input list *xs*. The inclusion of the invariance seems suspect when considering the implementation of the *fromList* function alone, since it has forced a more expensive implementation than the naive one. However, the assumptions allow the other operations to be implemented more efficiently.

Turning attention to the implementation of functions that construct deques, here is the function *empty*, where both components contain nothing:

```
empty :: Deque a
empty = Deque [] []
```

The symmetric nature of deques is seen in the way that the function *snoc* is implemented. While the following is a tempting implementation for its simplicity, it does not satisfy the representation invariant for deques:

```
snoc :: Deque a → a → Deque a
snoc (Deque xs sy) x = Deque xs (x : sy)
```

The function adds the element *x* onto the head of the list *sy*. Since the list *sy* is reversed, this effectively places *x* onto the end of the resulting list. The problem arises if *xs* is empty and *sy* is not: in this case the invariant is violated.

To check the invariance, evaluate *toList* (*snoc* (*fromList xs*) *x*).

One way to fix this problem is to add a special case for when *isEmpty xs* holds, which corresponds to a pattern match when the first argument is empty:

```
snoc :: Deque a → a → Deque a
snoc (Deque [] sy) x = Deque sy [x]
snoc (Deque xs sy) x = Deque xs (x : sy)
```

In the first case, when evaluating *snoc* (*Deque xs sy*) *x* in the case where *isEmpty xs*, the invariance ensures that *isSingle sy* holds, which validates shifting the position of *sy* to the first parameter.

Following these implementations, checking if a deque is empty is achieved by checking that both components are:

```
isEmpty :: Deque a → Bool
isEmpty (Deque xs sy) = isEmpty xs ∧ isEmpty sy
```

When it comes to singleton lists, there are two possibilities, depending on which component is empty:

```
isSingle :: Deque a → Bool
isSingle (Deque xs sy) = (isEmpty xs ∧ isSingle sy) ∨ (isSingle xs ∧ isEmpty sy)
```

Both of these functions are constant-time operations.

The representation is particularly interesting in the definition of *tail*. It is important that the remaining *Deque* is rebalanced in order to benefit from good complexity.

```
tail :: Deque a → Deque a
tail (Deque []       []) = error "tail: empty list"
tail (Deque []       sy) = empty
tail (Deque [x]      sy) = fromList (reverse sy)
tail (Deque (x : xs) sy) = Deque xs sy
```

Now consider the complexity of executing *tail xs* for a deque *xs* where $n = length\ xs$. When run in isolation, with no assumptions it should be clear that the cost is in the worst case is $O(n)$, where the case *Deque* [*x*] *sy* is encountered, since *fromList* and *reverse* must be used. However, a more careful analysis reveals that this cost is rarely incurred.

Consider, for example, the repeated application of *tail* in a chain, starting with xs_0 :: *Deque a*, where $n = length\ xs_0$.

$$xs_0 \overset{tail}{\rightsquigarrow} xs_1 \overset{tail}{\rightsquigarrow} xs_2 \overset{tail}{\rightsquigarrow} \ldots \overset{tail}{\rightsquigarrow} xs_n$$

The cost of these successive calls clearly reduces on each iteration. But not only that, two successive calls of *tail* will not invoke the costly *Deque* $[x]$ *sy* case: the result of *fromList* ensures that the list *sy* is split into two balanced parts. If the worst cases are all taken individually to be $O(n)$, then this will give too high an approximation to the overall cost.

10.2 Amortization

The complexity is an example of *amortised analysis*, where operations must be understood in a wider context, rather than treating them in isolation. The general setting is where there is a sequence of operations $op_0 \ldots op_n$ acting on an initial datastructure xs_0.

A comprehensive overview of amortized complexity in a functional setting is covered by Schoenmakers [1992]

$$xs_0 \overset{op_0}{\rightsquigarrow} xs_1 \overset{op_1}{\rightsquigarrow} \ldots \overset{op_n}{\rightsquigarrow} xs_{n+1}$$

To perform amortized analysis on these operations, there are three things to define:

1. A *cost function* $C_{op_i}(xs_i)$ for each operation op_i on data xs_i.
2. An *amortized cost function* $A_{op_i}(xs_i)$ for each operation op_i on data xs_i.
3. A *size function* $S(xs)$ that calculates the size of data xs.

The costs functions estimate how many steps it would take for each operation to execute.

The goal is to define these functions so that they can do an accounting of how much work needs to be done to execute an operation on a datastructure. They should be defined so that the following holds:

$$C_{op_i}(xs_i) \leqslant A_{op_i}(xs_i) + S(xs_i) - S(xs_{i+1}) \tag{10.1}$$

This says that for any given data xs_i, its cost of executing the operation op_i is less than the amortized cost, plus the difference between the datastructure before and after the operation.

If this inequality can be shown to be true, then the summation over a series of operations is given by:

$$\sum_{i=0}^{n-1} C_{op_i}(xs_i) \leqslant \sum_{i=0}^{n-1} A_{op_i}(xs_i) + S(xs_0) - S(xs_n)$$

Furthermore, when $S(xs_0) = 0$, then this implies:

$$\sum_{i=0}^{n-1} C_{op_i}(xs_i) \leqslant \sum_{i=0}^{n-1} A_{op_i}(xs_i) \tag{10.2}$$

This says that the sum of the cost functions is less than the sum of the amortized costs. For example, if $A_{op_i}(xs) = 1$, then the total cost is bounded by $O(n)$.

This technique is now demonstrated by calculating the cost of executing operations on a deque. First, various costs are assigned. For *cons*, *snoc*, *head*, and *last*, the cost is simply 1, since these can be performed by a simple pattern match:

$$\begin{aligned} C_{cons}(xs) &= 1 & C_{snoc}(xs) &= 1 \\ C_{head}(xs) &= 1 & C_{last}(xs) &= 1 \end{aligned}$$

The *tail* function is a more expensive operation: when *xs* is a singleton list it will cost as many steps as there are elements in the reversed:

$$C_{tail}(\mathit{Deque}\ xs\ sy) = \textbf{if}\ \mathit{length}\ xs > 1\ \textbf{then}\ 1\ \textbf{else}\ \mathit{length}\ sy$$

Now a simple amortized cost is given to all the operations.

$$A_{op}(xs) = 2$$

This cost is obviously higher than the real cost of some operations, and lower than the real cost of others.

Finally, a size function is assigned to the data:

$$S(\mathit{Deque}\ xs\ sy) = |\mathit{length}\ xs - \mathit{length}\ sy|$$

With these pieces in place, it is easy to verify that Equation 10.1 holds. Consider the situation $\mathit{Deque}\ xs'\ sy' = \mathit{tail}\ (\mathit{Deque}\ xs\ sy)$, where $\mathit{length}\ sy = k$. In the worst case, when *xs* is a singleton list, this implies that:

$$\begin{aligned} S(\mathit{Deque}\ xs\ sy) &= k - 1 \\ S(\mathit{Deque}\ xs'\ sy') &= 1 \end{aligned}$$

So, substituting into Equation 10.1 this results in:

$$\begin{aligned} C_{tail}(\mathit{Deque}\ xs\ sy) \leqslant A_{tail}(\mathit{Deque}\ xs\ sy) + S(\mathit{Deque}\ xs\ sy) - S(\mathit{Deque}\ xs'\ sy') \\ \iff \\ k \leqslant 2 + (k-1) - 1 \end{aligned}$$

This is clearly true. Therefore the time complexity of these instructions is bounded by $O(n)$, and the amortized cost of *tail* is $O(1)$.

11
Random Access Lists

God made the integers,
all else is the work of man.

Leopald Kronecker

This section explores a representation of lists that is inspired by numerical representations. These are the *random access* lists, which benefit from constant time consing and log time access to arbitrary elements.

11.1 Peano Numbers

Peano numbers are a simplistic way of counting natural numbers: a number is either zero, or one more than some other number. As data, this is represented by the *Peano* datatype. Here it is, along with a few primitive functions:

```
data Peano = Zero | Succ Peano

inc :: Peano → Peano
inc n = Succ n

dec :: Peano → Peano
dec (Succ n) = n

add :: Peano → Peano → Peano
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

Now notice how the structure of lists is essentially the same, except that there is now data involved:

```
data List a = Empty | Cons a (List a)

cons :: a → List a → List a
cons x xs = Cons x xs

tail :: List a → List a
tail (Cons x xs) = xs

(++) :: List a → List a → List a
Empty ++ ys       = ys
(Cons x xs) ++ ys = Cons x (xs ++ ys)
```

This similarity points to a deep connection between numbers and lists of a given length.

11.2 *Binary Numbers*

Now, a better counting system than Peano numbers is to use binary digits instead:

```
type Binary = [Digit]
data Digit = O | I
  deriving Eq
```

The idea is to use a list of digits in a least-significant-bit first representation, so that the list $[I, O, I, I]$ represents the number $13 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$.

Later *Digit* values will be compared, so **deriving** *Eq* is added to the end of the datatype declaration to allow this to happen.

The operations for adding and subtracting numbers perform binary arithmetic in the usual way:

```
inc :: Binary → Binary
inc []       = [I]
inc (O : bs) = I : bs
inc (I : bs) = O : (inc bs)
```

The complexity of *inc* is $O(n)$ in the worst case, where $n = \mathit{length}\ \mathit{bs}$, and *bs* contains only *I*. However, repeated applications of *inc* make it so that this case does not happen every time. Amortized analysis is a better way to analyse this complexity.

First, a cost is assigned for the *inc* instructions. This time, the cost can be calculated carefully:

$$C_{inc}(bs) = t + 1 \ \textbf{where} \ t = \mathit{length}\ (\mathit{takeWhile}\ (\equiv I)\ bs)$$

The cost of executing an *inc* instruction is the number of consecutive *I* values starting from the head.

For the amortized cost, a constant value of 2 is chosen:

$$A_{inc}(bs) = 2$$

Finally, the size function will count the number of *I* values there are in a given *bs*:

$$S_{inc}(bs) = b \ \textbf{where} \ b = \mathit{length}\ (\mathit{filter}\ (\equiv I)\ bs)$$

With these ingredients in place, it is time to apply them to the amortized cost equation (Equation 10.1) to see if the inequality can be satisfied.

Given a list of binary digits *bs* and another $bs' = \mathit{inc}\ \ bs$, the following holds:

$$\begin{aligned}
& C_{inc}(bs) \leqslant A_{inc}(bs) + S_{inc}(bs) - S_{inc}(bs') \\
\Leftrightarrow & \\
& t + 1 \leqslant 2 + b - b' \ \textbf{where} \ b' = b - t + 1 \\
\Leftrightarrow &
\end{aligned}$$

$$t + 1 \leqslant 2 + b - (b - t + 1)$$
$$\Leftrightarrow$$
$$t + 1 \leqslant t + 1$$

This is true, and so it is the case that an appropriate amortized cost for *incr* is $O(1)$.

11.3 Binary Tree Lookup

Balanced binary trees allow efficient access to their elements. Here is their datatype definition:

```
data Tree a = Tip | Leaf a | Fork Int (Tree a) (Tree a)
```

This datatype has three constructors. There are two base cases. One simply has the value *Tip*, which represents a tree with no data. The other base case has values such as *Leaf x*, containing only the element x. In the recursive case, *Fork n l r* puts together two trees l and r, and stores a value n which is the size of the tree, calculated as the number of values held in its leaves.

A *smart constructor* is used to maintain that the tree stores its size properly in n.

```
fork :: Tree a → Tree a → Tree a
fork l r = Fork (length l + length r) l r
```

This way, if new forks are constructed only with the *fork* function, then the resulting trees will be well-behaved.

In fact any *Tree a* can be viewed as a list, where a traversal of the leaves from left to right gives the correct sequence.

```
instance List Tree where
  toList :: Tree a → [a]
  toList (Tip)        = [ ]
  toList (Leaf x)     = [x]
  toList (Fork n l r) = toList l ⧺ toList r
```

The *length* function can then be used to extract the number of elements in the tree:

```
  length :: Tree a → Int
  length (Tip)        = 0
  length (Leaf x)     = 1
  length (Fork n l r) = n
```

Most of the other functions are fairly routine, but one that is of interest is the unsafe lookup function:

```
(!!) :: Tree a → Int → a
Tip !! n = error "(!!): no values in a Tip!"
Leaf x !! 0 = x
Fork n l r !! k
```

```
    | k < m      = l !! k
    | otherwise = r !! (k − m)
    where m = length l
```

If the tree is balanced, then this operation takes $O(\log n)$ time: each recursive call considers half of the remaining tree.

11.4 Random Access Lists

The standard representation of lists models itself on the simple counting system of Peano numbers. Peano numbers increase linearly in value as the size of their representation grows. Binary numbers however grow exponentially in value as the size of their representation grows. Random access lists model the structure of lists on binary numbers, and benefit from this fact.

The datatype for random access list is *RAList*:

```
newtype RAList a = RAList [Tree a]
```

An alternative definition is to use a *Tree′* where there is no *Tip* constructor. A *Tree′* can only represent nonempty lists and so it cannot fully implement the *List* class. However, the type *Maybe* (*Tree′ a*) is a list representation, where *Nothing* represents an empty list, and *Just t* represents some nonempty list encoded by *t*. Making this an instance requires either a type synonym or a way to compose type constructors.

It is possible to provide a *List RAList* instance with the required functions. Here are some of the more interesting definitions.

```
instance List RAList where
  toList :: RAList a → [a]
  toList (RAList ts) = (concat ∘ map toList) ts
```

Now the *RAList* will inherit much of the complexity of a *Tree a*. For instance, here is the definition of an unsafe lookup:

```
(!!) :: RAList a → Int → a
RAList (t : ts) !! k
    | isEmpty t = RAList ts !! k
    | k < m      = t !! k
    | otherwise  = RAList ts !! (k − m)
  where m = length t
```

Given $xs :: RAList\ a$, the cost of performing $xs\ !!\ k$ is $O(\log k)$ in the worst case.

The interesting operation is the *cons* function:

```
cons :: a → RAList a → RAList a
cons x xs = RAList (consTrees (Leaf x) xs)
  where
    consTrees :: Tree a → RAList a → [Tree a]
    consTrees t (RAList [])          = [t]
    consTrees t (RAList (Tip : ts)) = t : ts
    consTrees t (RAList (t′ : ts))   = Tip : consTrees (fork t t′) (RAList ts)
```

Notice that this follows the structure of the *inc* :: *Binary* → *Binary* function, therefore benefiting from similar amortized complexity.

12
Searching

> TRANIO: (as LUCENTIO) He is my father, sir, and sooth to say, In count'nance somewhat doth resemble you.
> BIONDELLO: (aside) As much as an apple doth an oyster, and all one.
>
> William Shakespeare
> Taming of the Shrew, 1590–1592

Searching for data is one of the most frequently executed operations. The operation can be performed on raw unsorted data, but this is only really desirable if a search is only ever performed once. More often than not there will be multiple queries and that is where some organisation becomes desirable. This chapter works towards binary search trees: a data structure that can be grown incrementally in an efficient way, and that allows efficient searching.

12.0.1 Equality

Searching for a value in a collection implies that there must be a way to check that the right value has been found. The easiest way to check is simply through equality. While it is clear that many things can be compared for equality, such as numbers, characters, and strings, this is not universally true: not all functions can be compared.

If all functions could be compared then the Halting problem would be solvable.

The *Eq* class is used to give the implementation of equality required. It says that a type a is an instance of the *Eq* class when the $(\equiv)$ function has an implementation:

```
class Eq a where
  (≡) :: a → a → Bool
```

Valid implementations of the *Eq* class should have an equality operator that behaves in an expected way. This is specified by giving the laws that make a valid equality operation. A valid $(\equiv)$ operation is one that is *reflexive*, *symmetric*, and *transitive*, where for all x, y, and

z:

$$
\begin{aligned}
x \equiv x & \qquad \text{(reflexivity)} \\
x \equiv y \Leftrightarrow y \equiv x & \qquad \text{(symmetry)} \\
x \equiv y \wedge y \equiv z \Rightarrow x \equiv z & \qquad \text{(transitivity)}
\end{aligned}
$$

These laws are an essential part of the specification that outline what can be expected of any implementation.

The *Prelude* version of *Eq* also includes negation ($\not\equiv$) which is not introduced here for brevity.

For instance, here is a definition of some custom datatype for fruit, and the implementation of *Eq Fruit* that demonstrates equality:

```
data Fruit = Apple | Orange
instance Eq Fruit where
  Apple  ≡ Apple  = True
  Orange ≡ Orange = True
  _      ≡ _      = False
```

This says that apples are apples and oranges are oranges, but nothing else holds true. The language does not enforce that the laws of equality hold for this instance, and it is up to the programmer to ensure that the implementation is valid.

12.1 Rummaging

Perhaps the simplest way to search for an element is to look through an entire collection. This idea can be modelled quite simply by a list of elements that have some means of comparing those elements for equality. The *rummage* function is used to find an element in a collection, where *rummage x xs* is true if and only if the element x is contained in xs.

```
rummage :: Eq a ⇒ a → [a] → Bool
rummage x [ ] = False
rummage x (y : ys)
  | x ≡ y     = True
  | otherwise = rummage x ys
```

This definition is a variation of my favourite joke: **if** *p x* **then** *True* **else** *False*.

The serious version is:

```
rummage x [ ]      = False
rummage x (y : ys) =
  x ≡ y ∨ rummage x ys
```

This avoids the repetition of *True*.

This function returns *False* if it reaches the empty list. Otherwise it compares the element x to y and will perform a short-circuiting disjunction with finding x in the remaining ys. In other words, if $x \equiv y$ then there is no need to evaluate recursively, but otherwise, the recursive call is made.

This function is as simple as it is inefficient. It is obvious that in the worst case it must check every element in a list of length n. A little thought will reveal that this is also the average case. That said, no preparation of the list was required before a *rummage*: this can be executed on any messy list. By placing elements in a datastructure with more order, it is possible to improve on this complexity.

12.2 *Ordered Lists*

One way to make searching faster is to strengthen the underlying assumption on the datastructure. Plausibly, things might be better if the elements are structured: then the searching could stop sooner rather than rummaging tirelessly through an unordered mess.

The assumption that the elements can be ordered is recorded with an *Ord* constraint, which expects the ($\leqslant$) relation to be defined. The *Ord* class itself relies on the existence of *Eq*, so that the order ($\leqslant$) acn be compatible with equality ($\equiv$). Thus, if a type has an *Ord* instance, it can be assumed that it also has an *Eq* instance. The opposite is not always true: two elements may be comparable for equality even though they cannot be ordered, like comparing apples and oranges.

Other operators such as ($<$), ($>$), and ($\geqslant$) are also part of the class in the *Prelude*. These can all be defined in terms of ($\leqslant$).

```
class Eq a ⇒ Ord a where
  (⩽) :: a → a → Bool
  ...
```

An appropriate definition of ($\leqslant$) is such that the relation is a *partial order*, which means that it is *reflexive*, *transitive*, and *antisymmetric*, where for all x, y, and z:

A relation that is reflexive and transitive is known as a *preorder*.

$$x \leqslant x \qquad \text{(reflexivity)}$$

$$x \leqslant y \wedge y \leqslant z \Rightarrow x \leqslant z \qquad \text{(transitivity)}$$

$$x \leqslant y \wedge y \leqslant x \Rightarrow x \equiv y \qquad \text{(antisymmetry)}$$

Notice that stipulating antisymmetry requires equality.

A variation of a partial order is a *total order* where the relation is also *connex*:

$$x \leqslant y \vee y \leqslant x \qquad \text{(connexity)}$$

Notice that connexity implies reflexivity, making connexity a weaker requirement: every total order is a partial order, but not the other way around.

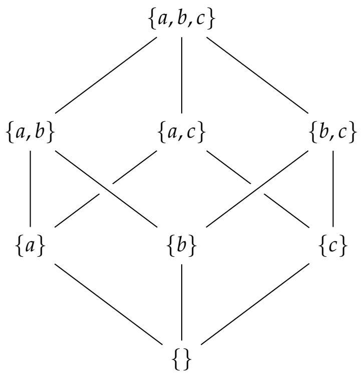


Figure 12.1: A Hasse diagram showing the subsets of $\{a, b, c\}$. The subset relation is a partial order.

One could imagine making a new class, *TOrd*, which is for a total order. It would have the same methods as *Ord*, but with the additional requirement of *connexity*. This is not generally implemented.

12.3 *Partially Ordered Set*

There are various ways to impose that the elements in a set have an ordering, and one way is to use a *partially ordered set*, also known as a *poset*.

```
class Poset poset where
  toPoset :: Ord a ⇒ [a] → poset a
  fromPoset :: poset a → [a]

  empty :: poset a
  insert :: Ord a ⇒ a → poset a → poset a
  delete :: Ord a ⇒ a → poset a → poset a

  member :: Ord a ⇒ a → poset a → Bool
  union :: Ord a ⇒ poset a → poset a → poset a
  inter :: Ord a ⇒ poset a → poset a → poset a
```

These operations are all intended to behave like the usual operations on sets. Since this encodes a set, the datastructure does not store duplicate elements.

Adding the *Ord* constraint alone is not enough to ensure a more efficient algorithm. The naive implementation of this interface is as an ordered list, where the *member* function is a more orderly version of *rummage*:

```
instance Poset [ ] where
  member :: Ord a ⇒ a → [a] → Bool
  member x [ ] = False
  member x (y : ys)
     | x ≡ y      = True
     | x < y      = member x ys
     | otherwise = False
```

Every good joke has many variations. Here is the serious version of *member*:

```
  member :: Ord a ⇒ a → [a] → Bool
  member x [ ]      = False
  member x (y : ys) =
     x ≡ y ∨ (x < y ∧ member x ys)
```

I find that this is a little cumbersome.

Alas, this code is barely any better. It is true that if the element is not in the list then this is likely to terminate more quickly. However, the time complexity remains the same.

12.4 Search Trees

The quicksort algorithm works by taking a pivot that is used to partition data into two parts: elements that are less than or equal to that pivot, and elements greater than it. The structure of this recursion can be captured in a *Tree*:

```
data Tree a = Nil | Node (Tree a) a (Tree a)
```

Creating such a tree from a list behaves much like the divide step of quicksort:

```
instance Poset Tree where
  toPoset :: Ord a ⇒ [a] → Tree a
  toPoset [ ]      = Nil
  toPoset (x : xs) = Node (toPoset us) x (toPoset vs)
    where (us, vs) = partition (⩽ x) xs
```

This partitions the list, which is done in $\Theta(n)$ time, and the hope is that the lists *us* and *vs* are each roughly of size $\frac{n}{2}$, leading to $2log(n)$ applications of *toPoset*.

Such trees are useful because when they are balanced they allow fast access to elements within the tree.

```
  member :: Ord a ⇒ a → Tree a → Bool
  member x Nil = False
  member x (Node lt y rt)
     | x ≡ y      = True
```

Somehow, this variation of the joke isn't that funny. Perhaps the punchline is too overworked. Also, the serious version is far too serious:

```
  member x (Node lt y rt) =
     x ≡ y ∨ (x < y ∧ member x lt) ∨ member x rt
```

```
    | x < y       = member x lt
    | otherwise = member x rt
```

If the tree is balanced, then its depth will force $T_{member}(n) \in O(log(n))$, where n is the number of elements in the tree.

However, the worst case complexity here is still linear. The fault is not with *member*, but with the construction of the *Tree* by *toPoset*: the hope of having lists of roughly equal size is easily dashed when the input is already sorted, for instance.

12.5 Binary Search Trees

Binary search trees, also known as AVL trees (named after Adelson-Velskii and Landis [1962]), are an interesting structure that carefully balance trees by keeping track of their height. The implementation is in terms of an *HTree*, where an extra parameter stores the height of the tree.

```
type Height = Int
data HTree a = HTip
             | HNode Height (HTree a) a (HTree a)
```

To ensure that the nodes are constructed in such a way that the height is properly preserved, a smart constructor can be used:

```
hnode :: HTree a → a → HTree a → HTree a
hnode lt x rt = HNode h lt x rt
  where
    h = (height lt ⊔ height rt) + 1
height :: HTree a → Int
height HTip = 0
height (HNode h lt x rt) = h
```

The interesting operation is *insert*, since this maintains that the tree is properly balanced. This makes use of *balancel* and *balancer* as smart constructors that create a new node whose subtrees are balanced.

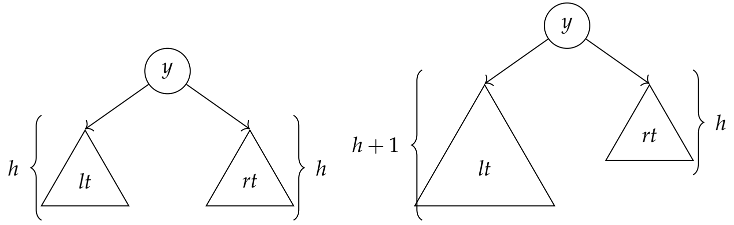
```
instance Poset HTree where
  insert :: Ord a ⇒ a → HTree a → HTree a
  insert x HTip = hnode HTip x HTip
  insert x t@(HNode _ lt y rt)
    | x ≡ y       = t
    | x < y       = balancel (insert x lt) y rt
    | otherwise = balancer lt y (insert x rt)
```

The invariance that is maintained by *balancel* and *balancer* is that the difference in height between any two siblings can be at most 1.

The definition proceeds by consdering the heights of the two trees *lt* and *rt* that are being passed as input. A simple case is where the height of *lt* and *rt* differ by at most 1 already. Focusing on *balancel*, where only the left tree has a value inserted into it, it is only necessary to compare *height lt* − *height rt*:

```
balancel :: HTree a → a → HTree a → HTree a
balancel lt y rt
  | height lt − height rt ⩽ 1 = hnode lt y rt
```

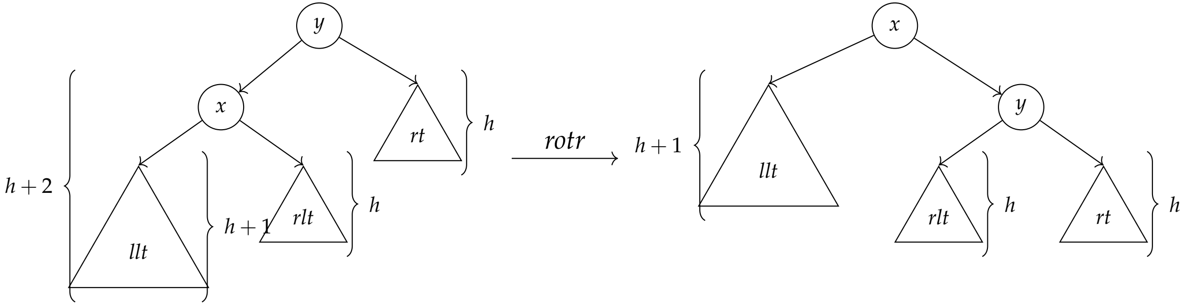
As a diagram, the tree must be one of the following two, which are already balanced:



Otherwise, assume that the difference is exactly 2: this is the case since the assumption is that the initial tree has an imbalance of at most 1 to begin with. Inserting a single extra element will at most increase this imbalance by an extra level.

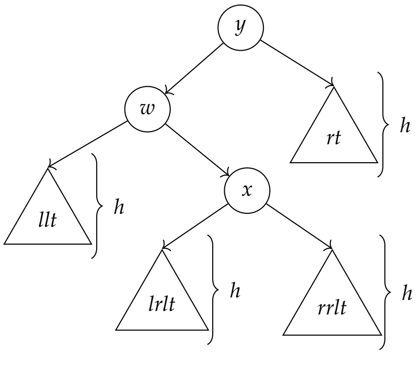
If *height lt* = *height rt* + 2 then further analysis on the subchildren of *lt* = *Node* _ *llt x rlt* yields some interesting cases.

Suppose that *height llt* > *height rlt*. Assuming that *height rt* = *h*, then *height lt* = *h* + 2, and *height llt* = *h* + 1. This is depicted by the left hand diagram below, and can be rotated to the right using *rotr* to produce the balanced tree to the right.

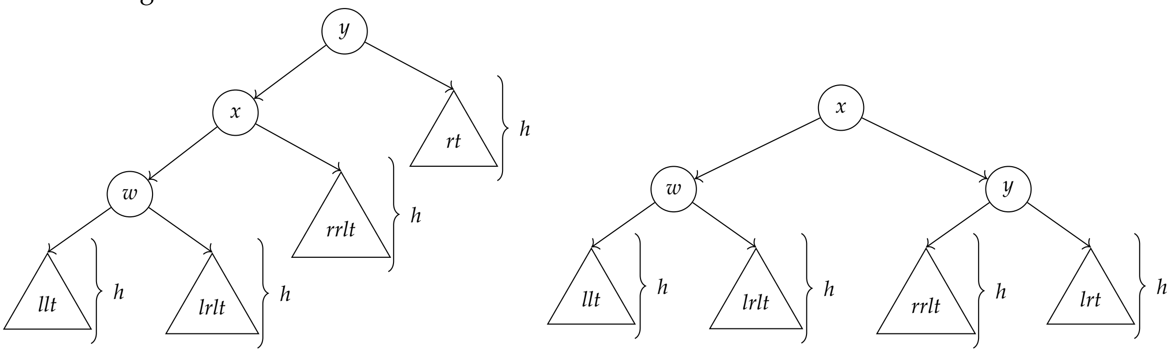


In fact, if *height llt* ≡ *height rlt* then this rotation also leaves the tree relatively balanced, where the bias is at most 1.

The final case is when *height llt* < *height rlt*. Assuming that *h* = *height rt*, it is possible to reason that the tree is shaped as follows, where *lt* = *Node* _ *llt w rlt* and *rlt* = *Node* _ *lrlt x rrlt*:



To balance this tree the goal will be to place x at the root, and have subtrees of equal height $h + 1$. This can be achieved by first rotating the left subtree *lt* to the left, and then rotating the whole result to the right.



These cases can be encoded by:

```
  | otherwise = case lt of
    HNode _ llt x rlt | height llt ⩾ height rlt → rotr (hnode lt y rt)
                      | otherwise                → rotr (hnode (rotl lt) y rt)
```

All that is needed is a suitable definition of *rotl* and *rotr*, which can be done by simple pattern matching:

```
rotr :: HTree a → HTree a
rotr (HNode _ (HNode _ p x q) y r) = hnode p x (hnode q y r)
rotl :: HTree a → HTree a
rotl (HNode _ p x (HNode _ q y r)) = hnode (hnode p x q) y r
```

The code for *balancer* follows a similar reasoning.

Turning to the analysis of complexity, it should be clear that *rotl*, *rotr* and *balancel* all take constant time: they all make use of contant time operations and do not recurse. Consequently, the *insert* function costs only $O(\log(n))$ time, assuming a balanced tree, to create a balanced tree with an element inserted.

13
Red-Black Trees

The one red leaf, the last of its clan,
That dances as often as dance it can,
Hanging so light, and hanging so high, On
the topmost twig that looks up at the sky.

Samuel Taylor Colleridge
Chritabel, 1797–1800

Red-black trees are another means of creating balanced trees. Unlike AVL trees, they do not need to store the height of the current tree, but work instead by storing a single bit that indicates the "colour" of a node: red or black.

The trees are represented quite simply as a modification of a binary tree with this additional information:

```
data Colour = R | B
data RBTree a = E
              | N Colour (RBTree a) a (RBTree a)
```

This datastructure is subject to two invariants:

1. Every red node most have a black parent node
2. Every path from the root node to a leaf must have the same number of black nodes

These conditions ensure that the tree is at most imbalanced by a factor of at most two in one of its branches.

As with other binary trees and their variations, the motivation for Red-Black trees is to provide fast searching, which is possible when the tree is (roughly) balanced.

The *insert* x function proceeds by inserting a new red leaf at the bottom of the tree that contains the element x. This is achieved by recursively calling the *go* function on the appropriate subtree until an empty node is found. Every node along the path to that leaf is balanced by applying the *balance* function. To ensure that the parent node is not red, the *blacken* function is applied to the final result.

```
insert :: Ord a ⇒ a → RBTree a → RBTree a
insert x t = blacken (go t)
```

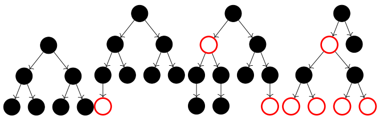


Figure 13.1: Valid Red-Black trees

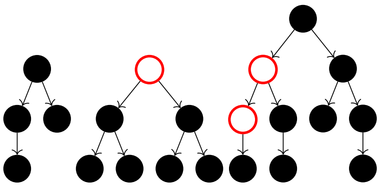


Figure 13.2: Invalid Red-Black trees

```
  where
    go :: RBTree a → RBTree a
    go E = N R E x E
    go t@(N c lt y rt)
       | x < y = balance c (go lt) y rt
       | x ≡ y = t
       | x > y = balance c lt y (go rt)
```

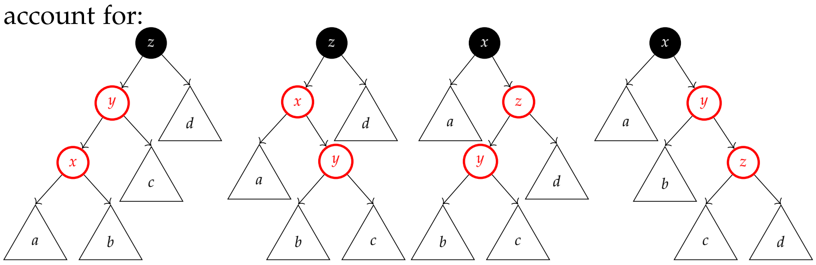
The *blacken* function simply changes the colour of a top node from *R* to *B*, and returns the original tree *t* otherwise:

```
blacken :: RBTree a → RBTree a
blacken (N R lt x rt) = N B lt x rt
blacken t             = t
```

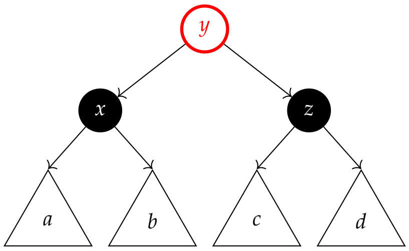
This function is only ever applied to the result of *go t*, which means that it will only affect the root node, thus enforcing the first invariant for that node.

The purpose of the *balance* function is to balance the tree by ensuring that there are no red nodes with red children. Assuming that the tree is valid to start with, the only new red node will have been inserted at one of the leaves. The innermost application of *balance* will be able to fix a potential red-red conlict, but may itself create a new red node that needs fixing by the next call to *balance*.

There are four different cases of interest that *balance* needs to account for:



These have been arranged so that the resulting tree that is required is always the following:



The implementation of *balance* requires encoding these trees using pattern matching:

```
balance :: Colour → RBTree a → a → RBTree a → RBTree a
balance B (N R (NR a x b) y c) z d  = N R (N B a x b) y (N B c z d)
balance B (N R a x (NR b y c)) z d  = N R (N B a x b) y (N B c z d)
balance B a x (N R (N R b y c) z d) = N R (N B a x b) y (N B c z d)
balance B a x (N R b y (N R c z d)) = N R (N B a x b) y (N B c z d)
balance c lt x rt                   = N c lt x rt
```

As *balance* is recursively called, the root note will eventually certainly be red, and so this is resolved with the *blacken* function.

14
Randomized Algorithms

> The assumption of an absolute determinism is the essential foundation of every scientific enquiry.
>
> Max Planck

A *randomized algorithm* is an algorithm that uses random values in its execution in order to produce a result. Such algorithms are desirable when they produce results quickly with a high probability. The performance characteristics of good randomized algorithms can be better than deterministic alternatives in different ways: they may use less time, less memory, or be able to cope with datasets that are much larger or that have difficult edge cases.

There are two broad classifications of randomized algorithms: *Monte Carlo* algorithms have a predictable running time but unpredictably compute a correct result, and *Las Vegas* algorithms have an unpredictable running time, but predictably compute a correct result.

The names have to do with the way that different casinos are run, but the following mnemonic seems more useful to remember the distinction: Monte Carlo is Maybe Correct and Las Vegas running Length Varies.

14.1 Determinism and Randomization

Functions always map the same inputs to the same outputs. This is known as Leibniz's law, or the *identity of indiscernibles*:

$$x = y \Rightarrow f\ x = f\ y$$

This holds true for all values x, y and functions f. The value of the output depends on the value of the input, and nothing else. A consequence is that no function can return a truly random result: every execution of a function on the same input will return the same result.

They can, however, exhibit pseudo-random behaviour by depending on some input that varies either explicitly or implicitly. Since truly random values cannot be generated, from now on, the term *random* will be used to mean *pseudo-random*.

The key idea behind random value generation is to start with a seed value from which a random value and a new seed can be extracted.

Random numbers are provided by the `random` package on Hackage. It can be installed with the command: `cabal install --lib random`

Seed values have type *StdGen* and can be created with the *mkStdGen* function:

$$mkStdGen :: Int \to StdGen$$

Once this value has been created, it can be passed to the *random* function, which will return a random value of a given type, along with a new seed of type *StdGen*. Here is the signature if the value desired is an *Int*:

$$random :: StdGen \to (Int, StdGen)$$

The new seed can then be passed on to the next invocation of *random* until all the random values that are desired have been extracted.

As an example of using this, here is the function *randoms*, which uses this instance to generate a list of random *Int*s from a given seed:

```
randoms :: StdGen → [Int]
randoms seed = x : randoms seed'
  where (x, seed') = random seed
```

This produces an infinite list of random values.

It is not possible to generate random values of every type automatically. However, the *Random* type class provides an interface for generating random values where an instance is available. Here is an idealized version of the class:

```
class Random a where
  random   :: StdGen → (a, StdGen)
  randoms  :: StdGen → [a]

  randomR  :: (a, a) → StdGen → (a, StdGen)
  randomRs :: (a, a) → StdGen → [a]
```

You can import the *Random* class from the `random` package with:

import *System.Random*

For simplicity, the version presented in these notes is a specialised version of *Random*: in its full generality the behaviour of *StdGen* is specified by another type class, and other functions are provided that are not relevant to this course.

The *random* function can be used to generate a single random value of some type *a*, along with a new *StdGen* ready for use again. A variant of this is *randomR* where random values are generated between a given range. Yet more variations are *randoms* and *randomR*, which generate an infinite list of random values, but do not return a new seed.

14.2 Randomized π

A classic Monte Carlo algorithm, though not one that is terribly efficient, deals with computing the value of π. The key observation is that a circle with radius 1 has an area of π. If a series of points are randomly chosen within the square enclosing that circle, then they will be contained in the circle with a ratio approaching $\pi/4$. After throwing 100000 random values at the unit circle the estimate of π is merely 3.14612.

This is a similar technique to Buffon's needle, developed in the 18th Century.

Using the *Random* interface, this can be encoded as follows:

```
montePi :: Double
montePi = loop (mkStdGen 42) samples 0
  where
    loop :: StdGen → Int → Int → Double
    loop seed 0 m = 4 × fromIntegral m / fromIntegral samples
    loop seed n m =
      let (x, seed′)  = randomR (0, 1) seed
          (y, seed″) = randomR (0, 1) seed′
          m′ = if inside (x, y) then m + 1 else m
          n′  = n − 1
      in loop seed″ n′ m′
samples :: Int
samples = 10000
```

The important point to note here is that the variables *seed*, *seed′*, and *seed″* must be carefully scheduled to happen sequentially. The first *seed* is used to generate *x*, the second *seed′* is used to generate *y*, and the third *seed″* is fed into the recursive body of *loop*.

Apart from the generation of the random values *x* and *y*, *montePi* uses the *inside* function, which checks whether the given coordinates lie inside a circle.

```
inside :: (Double, Double) → Bool
inside (x, y) = x × x + y × y ⩽ 1
```

This is where the real work happens. Values that are inside the unit circle increment the value *m*, and At each iteration the *n* parameter of the *loop* gets closer to 0, at which point the proportion between the values *m* that are inside the circle and the total number of samples is used to approximate π.

14.3 Sequencing Random Generators

Threading seeds around is somewhat tedious and error prone: if the same seed is used more than once then the values that are supposed to be independent random variables will end up being the same value.

One way to resolve this is to handle the seed generation automatically. This can be achieved in a context *m* which supports the sequential generation of random values. In this variation the generation and threading of *seed* values is left completely implicit. The key change in the following code is the use of the **do** keyword, which indicates that the following block of code is to be executed sequentially, one line at a time:

```
montePi′ :: MonadRandom m ⇒ m Double
montePi′ = loop samples 0
  where
    loop :: MonadRandom m ⇒ Int → Int → m Double
    loop 0 m = return (4 × fromIntegral m / fromIntegral samples)
    loop n m = do
      x ← getRandomR (0,1)
      y ← getRandomR (0,1)
      let m′ = if inside (x,y) then m + 1 else m
          n′ = n − 1
      loop n′ m′
```

Notice that the value in the base case is wrapped around by a *return*, and the assignment of the values x and y is through special notation that indicates that they are the result of a sequential operation *getRandomR* (0, 1). The type of this function is:

```
getRandomR :: MonadRandom m ⇒ (Int, Int) → m Int
```

Just like its pure counterpart *randomR*, it takes a pair that indicates the range of the values it should produce. However, no seed is required or delivered: this is managed behind the scenes automatically.

The *getRandomR* function is part of the *MonadRandom* class:

```
class Monad m ⇒ MonadRandom m where
  getRandom   :: Random a ⇒ m a
  getRandoms  :: Random a ⇒ m [a]
  getRandomR  :: Random a ⇒ (a,a) → m a
  getRandomRs :: Random a ⇒ (a,a) → m [a]
```

This interface says that there are different contexts where it is possible handle the sequential generation of random values.

One way to evaluate this program is to use *evalRand*:

```
evalRand :: Rand StdGen a → StdGen → a
```

This forces the type m contrained by *MonadRandom m* to be *Rand StdGen*. Using this, the following line evaluates a (very bad) approximation to π:

```
GHCi > evalRand montePi′ (mkStdGen 42)
3.1264
```

This invocation makes the first parameter be *montePi′* :: *Rand StdGen Double*, and the value of the seed is *mkstDgen* 42 :: *StdGen*. The result is of type *Double*. Since this is a pure function it will always return the same value.

Another way to make use of the *IO* context, where *montePi′* :: *IO Double*. The program *montePi′* is the same as before, but its type has been resolved differently.

```
printPi :: IO ()
printPi = do pi ← montePi′
             print pi
```

This assigns the value of executing *montePi′* to *pi* :: *Double*, which is then printed. Here, the system provides the random values, rather than a fixed *StdGen*. Different runs of this program will return (slightly) different results:

```
GHCi > printPi
3.1208
GHCi > printPi
3.1432
```

This is due to the fact that the seed that is being passed around is being implicitly updated between executions.

14.4 Random Streams

A different approach to solving this problem is to use the *randomRs* function. In this version, all of the random values are generated before being transformed into an appropriate sample:

```
montePi'' :: Double
montePi'' = 4 × fromIntegral (length (filter inside xys)) / fromIntegral samples
  where xys = take samples (pairs (randomRs (0, 1) (mkStdGen 42) :: [Double]))
pairs :: [a] → [(a, a)]
pairs (x : y : xys) = (x, y) : pairs xys
```

This solution is quite neat, but is somewhat different in character to the more imperative implementation of *montePi*: it has avoided the problem of having to thread the *seed* values around. One criticism of the function is that the value *mkStdGen* 42 is hard coded into the implementation. An obvious way of changing this is to make *seed* a parameter to this function, and to pass *mkStdGen* 42 as an argument.

An alternative to making a new parameter is to change the program to use a *MonadRandom* constraint.

```
montePi''' :: MonadRandom m ⇒ m Double
montePi''' = do
  rxys ← getRandomRs (0, 1)
  let xys = take samples (pairs (rxys))
  return (4 × fromIntegral (length (filter inside xys)) / fromIntegral samples)
```

Just as with the previous example, this can now be used flexibly, either using *evalRand* where the seed is provided as an argument, or in a context such as *IO*.

15 Treaps

> Quantum theory yields much, but hardly brings us close to the Old One's secrets. I, in any case, am convinced He does not play dice with the universe.
>
> Albert Einstein, 1926

15.1 Treaps

The binary search trees that have been studied so far have been very carefully balanced by looking at information that is stored in nodes and performing rotations to impose desired properties.

An alternative approach is to use randomization to ensure that the tree is balanced on construction. A *treap* structure, which is the combination of a binary *tr*ee and a h*eap*. The values that are stored in a treap are in symmetric order, so that compared to the value at a node, values to the left are smaller, and values to the right are larger. Additionally, parent nodes have a higher priority in the heap than their children.

```
data Treap a = Empty | Node (Treap a) a Int (Treap a)
  deriving Show
```

Here a value *Node l v p r* holds a left child *l*, a value *v*, a priority *p*, and a right child *r*.

This allows for an efficient *member* function, since values can be compared and are stored in order:

```
member :: Ord a ⇒ a → Treap a → Bool
member x Empty = False
member x (Node a y _ b)
  | x < y = member x a
  | x ≡ y = True
  | x > y = member x b
```

The insertion function into this data structure takes a value and a priority and ensures not only that the node is inserted into the tree in symmetric order, but also that the heap property on priorities is maintained.

```
insert :: Ord a ⇒ a → Int → Treap a → Treap a
insert x p Empty = Node Empty x p Empty
insert x p (Node a y q b)
    | x < y  = lnode (insert x p a) y q b
    | x ≡ y  = Node a y q b
    | x > y  = rnode a y q (insert x p b)
```

Notice that this *insert* function is quite different to the usual interface: it makes use of a priority *p*, which will help decide where elements should be positioned. Later, randomized treaps will be introduced to remove this requirement (Section 15.2).

The smart constructors for this structure are for when there is an insertion to the left or to the right.

```
lnode :: Treap a → a → Int → Treap a → Treap a
lnode Empty y q c = Node Empty y q c
lnode l@(Node a x p b) y q c
    | q ⩽ p      = Node l y q c    -- = Node (Node a x p b) y q c
    | otherwise = Node a x p (Node b y q c)
rnode :: Treap a → a → Int → Treap a → Treap a
rnode a x p Empty = Node a x p Empty
rnode a x p r@(Node b y q c)
    | p ⩽ q      = Node a x p r    -- = Node a x p (Node b y q c)
    | otherwise = Node (Node a x p b) y q c
```

To delete a node, the tree is recursively descended until the appropriate node is found. At this point, the value in the node is discarded and its two subnodes are merged.

```
delete :: Ord a ⇒ a → Treap a → Treap a
delete x Empty = Empty
delete x (Node a y q b)
    | x < y =  Node (delete x a) y q b
    | x ≡ y = merge a b
    | x > y =  Node a y q (delete x b)
```

This makes use of a function that can merge treaps, while maintaining their properties:

```
merge :: Treap a → Treap a → Treap a
merge Empty r = r
merge l Empty = l
merge l@(Node a x p b) r@(Node c y q d)
    | p < q =       Node a x p (merge b r)
    | otherwise = Node (merge l c) y q d
```

As with other treelike structures, it is possible to convert it to a list:

```
toList Empty = [ ]
toList (Node a x p b) = toList a ++ [x] ++ toList b
```

As usual, this is improved by replacing append with function composition, which results in a version that uses an accumulating parameter.

```
toList :: Treap a → [a]
toList t = toList′ t [ ]
  where
    toList′ :: Treap a → [a] → [a]
    toList′ Empty xs = xs
    toList′ (Node a x p b) xs = toList′ a (x : (toList′ b xs))
```

This version turns a treap into a list in linear time.

Making a *Treap* from a list requires values to come paired with their priorities. If the priorities are randomly distributed, then on average the resulting tree will be balanced.

```
fromList :: Ord a ⇒ [a] → Treap a
fromList xs = foldr (uncurry insert) Empty (zip xs (randoms seed))
  where seed = mkStdGen 42
```

When a *Treap* is constructed in this way the *member* function will take $\log n$ time on average, where n is the number of elements in the treap.

15.2 Randomized Treaps

The *Treap* datastructure has a specialised *insert* function that requires a priority to be presented along with the value to be inserted. Although this technically works, it is somewhat inconvenient that the *insert* function requires priorities to be given explicitly. A better approach is to embed the random variables in the treap itself and allow those priorities to be generated.

A *randomized treap* is a treap whose priorities are independent and uniformly distributed continuous random variables. Such variables can be extracted from a standard random number generator, which is abstracted by values of the type *StdGen*. The random variable generator is stored alongside an ordinary treap.

```
data RTreap a = RTreap StdGen (Treap a)
```

The *StdGen* can be used to create a random value whenever *random* :: *StdGen* → (*Int*, *StdGen*) is called, where *random seed* returns a pair $(x, seed')$ such that x is a uniformly distributed value, and $seed'$ is the next random generator.

The $insert'$ will therefore use the number generator to create a new priority for each insertion, and update the generator in the structure accordingly:

```
insert′ :: Ord a ⇒ a → RTreap a → RTreap a
insert′ x (RTreap seed t) = RTreap seed′ (insert x p t)
  where (p, seed′) = random seed
```

All that is left is the instantiation of a new randomized treap. This can be done in the empty constructor. As a pure function there needs to be a deterministic seed:

```
empty′ :: RTreap a
empty′ = RTreap (mkStdGen 42) Empty
```

This will, of course, return the same tree given the same input. To allow for a different tree at each instance requires the randomness to be threaded through:

```
empty″ :: StdGen → RTreap a
empty″ seed = RTreap seed Empty
```

As usual, a *fromList* function can be created for randomized treaps, so long as the elements can be ordered.

```
fromList′ :: Ord a ⇒ [a] → RTreap a
fromList′ xs = foldr insert′ empty′ xs
```

The *toList′* function is essentially the same as for *Treap*s:

```
toList′ :: RTreap a → [a]
toList′ (RTreap seed t) = toList t
```

Notice that *seed* is not required to build the new list.

15.3 Randomized Quicksort

The standard deterministic quicksort has an average complexity of $O(n \log n)$, but a terrible worst case of $O(n^2)$.

Randomized treaps can be used to create a randomized quicksort. The function *rquicksort* below uses a fixed seed, since *fromList′* uses *empty′*.

Beware: randomization of quicksort can happen in two different places: by randomizing the input list order and by randomizing the pivots. Randomizing the input list order does not help with the expected worst case of the input consisting of the same repeated element.

```
rquicksort :: Ord a ⇒ [a] → [a]
rquicksort xs = toList′ (fromList′ xs)
```

Now the worst case *expected* runtime is $O(n \log n)$.

16 Randomized Binary Search Trees

A *randomized binary search tree* behaves like an ordinary binary search tree most of the time, but with some probability will insert a value at its root. The underlying datatype is an ordinary binary tree:

```
data BTree a = BNil
             | BNode (BTree a) a (BTree a)
```

Randomized binary search trees make use of two insertion functions. One is the ordinary insertion that places an element in its position amongst siblings:

```
insert :: Ord a ⇒ a → BTree a → BTree a
insert x BNil = BNode BNil x BNil
insert x (BNode l y r)
  | x < y = BNode (insert x l) y r
  | x ≡ y = BNode l y r
  | x > y = BNode l y (insert x r)
```

The other insertion places an element at the root, and then adjusts the tree accordingly with rotations:

```
insertRoot :: Ord a ⇒ a → BTree a → BTree a
insertRoot x BNil = BNode BNil x BNil
insertRoot x (BNode l y r)
  | x < y = rotr (insertRoot x l) y r
  | x ≡ y = BNode l y r
  | x > y = rotl l y (insertRoot x r)

rotr :: BTree a → a → BTree a → BTree a
rotr (BNode a x b) y c = BNode a x (BNode b y c)

rotl :: BTree a → a → BTree a → BTree a
rotl a x (BNode b y c) = BNode (BNode a x b) y c
```

So far there has been nothing special in this construction. A *randomized binary search tree* contains a random seed and keeps track of the number of elements in the tree:

```
data RBTree a = RBTree StdGen Int (BTree a)
```

The empty *RBTree* simply instantiates the seed. As usual, this can be done implicitly with a default seed:

```
empty :: RBTree a
empty = RBTree (mkStdGen 42) 0 BNil
```

Now the interesting part. The *insert′* will insert an element into a tree with n elements in the ordinary way, but with probability $\frac{1}{n+1}$ will instead insert the value at the root.

```
insert′ :: Ord a ⇒ a → RBTree a → RBTree a
insert′ x (RBTree seed n t)
        | p ≡ 0      = RBTree seed′ (n + 1) (insertRoot x t)
        | otherwise = RBTree seed′ (n + 1) (insert x t)
    where
        (p, seed′) = randomR (0, n) seed
```

This maintains balance with a very high probability, but note, only returns correct results when distinct elements inserted using this function at most once.

17 Mutable Datastructures

Although it often leads to code that is difficult to analyse, the ability to mutate state can be desirable. This chapter explores mutable datastructures, and how they can be expressed in Haskell. The main motivation for using a mutable datastructure is that a location of memory can be reused, thus requiring less garbage collection which will in turn open up the possibility of more efficiency.

17.1 Mutable References

The *fib* function will once again serve as an example. This time, the goal is to write a version that keeps track of the value of the previous two Fibonacci numbers in the sequence.

Writing this as a pure function is as follows:

```
fib :: Int → Integer
fib n = loop n 0 1
  where
    loop 0 x y = x
    loop n x y = loop (n − 1) y (x + y)
```

The result of *loop n x y* is *x* when the base case of $n = 0$ is reached. Otherwise, the loop is called again where *n* is decremented, and the two parameters are updated to contain the next two values of the sequence. The loop is initialized with values 0 and 1.

A different way to implement this function is to use mutable state. A mutable value of type *STRef s a* holds a mutable reference to *a* that can be created, read, and modified with three primitive functions:

```
newSTRef   :: a → ST s (STRef s a)
readSTREf  :: STRef s a → ST s a
writeSTRef :: STRef s a → a → ST s ()
```

Notice that these operations all return values that work within an *ST s* context.

Such values can only be extracted with the *runST* function:

```
runST :: (forall s ∘ ST s a) → a
```

The *runST* function is rather special in that it processes a sequential computation, but remains a pure value by preventing any of its internal state from escaping to the outside world.

The exact details of how this works are beyond the scope of this course and it is enough to know that the first argument of type *ST s a* represents a computation that results in a value of type *a*. Since the scope of *s* is restricted, it is not allowed to appear in the type *a*, and so types tagged with *s* cannot appear as the result of a computation. In fact, the type of *ays* :: *STArray s Int Bool* reveals this *s* within the computation: it would be a type error to try to return *ays*.

By way of example, here is how *fib* could be implemented in an imperative style with the *STRef* operations:

```
fib′ :: Int → Integer
fib′ n = runST $ do
  rx ← newSTRef 0
  ry ← newSTRef 1
  let loop 0 = do
        x ← readSTRef rx
        return x
      loop n = do
        x ← readSTRef rx
        y ← readSTRef ry
        writeSTRef rx y
        writeSTRef ry (x + y)
        loop (n − 1)
  loop n
```

Notice that this code all sits inside a *runST* clause. The **do** keyword indicates that the code that follows should be executed in sequence, one line after the other. The definition of *loop* reveals that in the base case the value *x* is returned. This is the value that is extracted by *runST*.

17.2 A Checklist

An essential mutable structure is the array. Just as with mutable references, thre are primitive operations for arrays too:

```
newArray   :: Ix i ⇒ (i, i) → a → ST s (STArray s i a)
readArray  :: Ix i ⇒ STArray s i a → i → ST a
writeArray :: Ix i ⇒ STArray s i a → i → a → ST s (STArray s i a)
```

These operations are all assumed to take constant time. In practice, this is only true for arrays that can fit into memory.

To see this in action, suppose the goal is to look for the smallest natural number that does not occur in a list of numbers. Here is a good specification:

```
minfree :: [Int] → Int
minfree xs = head ([0 ..] \\ xs)
```

Bird [2010] presents a beautiful functional solution to this that uses divide and conquer and still has linear complexity.

```
(\\) :: Eq a ⇒ [a] → [a] → [a]
us \\ vs = filter (¬ ∘ flip elem vs) us
```

This specifies the problem nicely, but it is not efficient.

One way to solve this problem efficiently is to keep a checklist of what has been seen, and to go through that checklist and find the first missing number.

```
minfree′ :: [Int] → Int
minfree′ xs = length (takeWhile id (checklist xs))
```

This makes use of the *checklist* function:

```
checklist :: [Int] → [Bool]
```

The idea is that this function will return a list of booleans, where the value at index x is *True* if x is a member of *xs*, and *False* otherwise.

Implementing *checklist* with a pure datastructure is a difficult task, but is quite standard as an imperative program. In the following, the lines following the **do** notation are to be read sequentially.

```
checklist xs = runST $ do
  ays ← newArray (0, m − 1) False :: ST s (STArray s Int Bool)
  sequence [writeArray ays x True | x ← xs, x < m]
  getElems ays
where
  m = length xs
```

This particular algorithm relies on mutability. To indicate this the main body of the code is wrapped within *runST*, which indicates that the code that follows is to be executed sequentially. Here, *ays* is an array of m booleans that stores the numbers that have been seen. Initially those booleans are all *False*. The list is *xs* then used to to create a list of *writeArray* commands, that sets the elmeent with index x in the array *ays* to *True*. These commands are executed in sequence using the *sequence* function. Finally, the elements in the array are returned using *getElems*.

17.3 Hashing

The ability to hash a value is a cornerstone of imperative programs. A hash is simply an *Int* that is associated to a value. To do this generically, here is the *Hashable* type class, that allows *hash* functions to be defined for different types:

```
class Hashable a where
  hash :: a → Int
```

For instance, *hash* `"Hello World!"` $= -17518273135331410753$. Typically hashes are used as indices into an array that contains elements that need later retrieval. Of course, storing an array with the whole range of *Int* as indicies is far too large even if if adjustments were made for negative numbers. Usually, a much smaller index space is created by working with the hash modulo the size of the array.

As an example, consider the *nub* function, which takes a list and removes duplicates. The standard immutable version takes $O(n^2)$ time where n is the length of the list. Using a mutable array, and hashing it is possible to bring this down to $O(n)$ if there are not too many hash collisions.

The following code builds a new array of size 256, where each entry is a list of values that have been seen. These lists are extracted using *runST*, and *concat* puts them together into a single list.

A world of frustration occurs is that *concat* ∘ *runST* cannot be written. The problem is that type inference for function composition is unable to resolve the type of *s* appropriately. Resolving this with a type annotation is beyond the scope of this course.

```
nub :: (Hashable a, Eq a) ⇒ [a] → [a]
nub xs = concat (runST $ do
  axss ← newListArray (0, 255) (replicate 256 [ ]) :: ST s (STArray s Int [a])
  sequence [do let hx = hash x ‘mod‘ 255
               xs ← readArray axss hx
               unless (x ∈ xs) $ do
                 writeArray axss hx (x : xs)
            | x ← xs]
  getElems axss)
```

The initial state of *axss* is given by *replicate* 256 [ ], which creates 256 empty lists. Then, a sequence of instructions is executed, where for each value x in *xs*, the hash modulo 266 of x is calculated and assigned to *hx*. The list *xs* stored in the array at that index is extracted. Unless x is an element of *xs*, the array is updated so that the list at the index *hx* also includes x. The hope is that *xs* is a small list and that this operation costs $O(1)$. Finally, the elements are extracted with *getElems*.

Although hashing is a useful technique, it has a downfall: the performance of algorithms that rely on it is prone to degrade if the hash space is too small for the entries that are using it. In the algorithm above, if there 256 buckets contains a list of size m, then the ‘elem’ function will take $O(m)$, which negates the benefit of the constant lookup.

17.4 Quicksort

One valid complaint about the quicksort algorithm previously discussed is that it does not describe an in-place algorithm. Indeed, one of the key features of quicksort is that it is possible to perform the sorting in a single array by swapping elements.

Swapping elements at give indicies is quite an elementary part of many in-place algorithms. This can be implemented with the *swap* function:

```
swap :: STArray s Int a → Int → Int → ST s ()
swap axs i j = do
  x ← readArray axs i
```

```
    y ← readArray axs j
    writeArray axs i y
    writeArray axs j x
```

Here the mutable array *axs* is taken as a parameter, as well as two indices *i* and *j*. The variables *x* and *y* are read from the array, and a written into the swapped positions.

The *qsort* function takes in a list *xs* and sets up the array *axs* with the appropriate size. This is then fed into the function *aqsort* which does the real work.

```
qsort :: Ord a ⇒ [a] → [a]
qsort xs = runST $ do
  axs ← newListArray (0, n) xs
  aqsort axs 0 n
  getElems axs
where n = length xs − 1
```

The fact that there is mutation happening is hidden from the rest of th system by wrapping sequential steps within a *runST*.

The *aqsort* function is relatively simple: it takes in the array *axs* as an argument as well as the two *i* and *j* that indicate the range of values that should be sorted.

```
aqsort :: Ord a ⇒ STArray s Int a → Int → Int → ST s ()
aqsort axs i j
   | i ⩾ j       = return ()
   | otherwise = do
     k ← apartition axs i j
     aqsort axs i (k − 1)
     aqsort axs (k + 1) j
```

When $i \geqslant j$ then this represents either a singleton or no value, in which case the work is complete. Otherwise, the *apartition* function is called that does the real work of partitioning the array between *i* and *j*. It returns some index *k* which indicates the index of the pivot that was chosen. The *aqsort* function is then recursively called on the two partitions on either side of *k*.

The *apartition* function is used to work with the mutable array and perform the partitioning in place. Calling *apartition axs p q* will partition the values in the array *axs* between the indexes *p* and *q*, using the pivot at index *p* as the pivot.

```
apartition :: Ord a ⇒ STArray s Int a → Int → Int → ST s Int
apartition axs p q = do
  x ← readArray axs p
  let loop i j
      | i > j = do swap axs p j
                   return j
      | otherwise = do
        u ← readArray axs i
      if u < x
        then do loop (i + 1) j
        else do swap axs i j
                loop i (j − 1)
  loop (p + 1) q
```

This code works by first extracting the value x at the pivot p. Then, the body of a loop is defined where *loop i j* will partition the elements between i and j using the pivot x located at p. The loop is executed with *loop* $(p+1)$ q, since the element at p is the pivot.

In the loop, if $i > j$ then there are no more elements to partition, and the correct position for x will be at the index j. Otherwise, the idea is that the loop will compare x to the element at index i. If the element is already in the right partition then the loop is called again with $i + 1$. If the the element at i is out of place, it is swapped with the element at j. Since the element now at j is in the correct partition, the loop continues with $j - 1$.

To finish off, Here is a version of the function that does the partitioning on ordinary lists. The twist is that it does so by first converting the list into an array, before doing an in-place partitioning by calling *apartition*.

```
partition :: Ord a ⇒ [a] → [a]
partition [ ] = [ ]
partition xs = runST $ do
  axs ← newListArray (0, n) xs
  apartition axs 0 n
  getElems axs
where n = length xs − 1
```

When the list is not empty, this works by creating a new array *axs* from the list *xs*, with indices ranging from 0 to n, where $n = \mathit{length}\ \mathit{xs} - 1$. The call to *apartition* is used to partition the array *axs*. Finally, when this is finished, the elements of the array are extracted using *getElems*, which will return them as a list.

17.5 Array Resizing

Arrays provide constant time access to their elements, but this comes at the cost of being fixed at a given size. That said, it is possible to provide behaviour that makes an array behave like a list of arbitrary length. The idea is to insert elements into the array until it is full. At this point a new array is created that is double the size of the old. Elements are copied into the new array, and processing continues. This is a good exercise in in amortized complexity: so long as the resizing operation is not called too frequently, the array continues to promise amortized constant time access to its elements.

Here is an implementation.

```
data ArrayList s a = ArrayList (STRef s Int) (STRef s Int) (STRef s (STArray s Int a))
```

The value *ArrayList n m axs* represents a list with *n* elements stored in the array *axs*, with a maximum capacity *m*.

Using the *ArrayList* constructor, here is empty array list with an array of some minimum size and uninitialized values, which is possible with the *newArray_* function:

```
newArray_ :: Ix i ⇒ (i, i) → ST s (STArray s i a)
```

Working with uninitialized variables is of course a dangerous business but in this case the invariance forces the array to be considered as empty.

For the empty *ArrayList*, there is an arbitrary initial capacity, set in this case to 8.

```
empty :: ST s (ArrayList s a)
empty = do pn   ← newSTRef 0
           pm   ← newSTRef m
           axs  ← newArray_ (0, m − 1)
           paxs ← newSTRef axs
           return (ArrayList pn pm paxs)
  where m = 8
```

The idea is that the array is to be filled starting from its highest index.

To extract the list that is being represented, the *toList* function must work within an *ST s* context.

```
toList :: ArrayList s a → ST s [a]
toList (ArrayList rn rm raxs) = do
  n ← readSTRef rn
  m ← readSTRef rm
  axs ← readSTRef raxs
  sequence [readArray axs i | i ← [m − n . . m − 1]]
```

The vaule *n* gives the number of elements to extract, and *m* shows the maximum capacity. Together these give the range of indices where values can be safely extracted from the array.

To insert values into the array requires some care. When the array is full, a new one is created with twice the previous capacity that contains all the old values.

```
insert :: a → (ArrayList s a) → ST s ()
insert x (ArrayList pn pm paxs) = do
  n   ← readSTRef pn
  m   ← readSTRef pm
  axs ← readSTRef paxs
  writeSTRef pn (n + 1)
  if n < m
    then do
      writeArray axs (m − n − 1) x
    else do
      let m′ = 2 × m
      writeSTRef pm m'
      axs' ← newArray_ (0, m' − 1)
      writeSTRef paxs axs'
      sequence [do x' ← readArray axs i
                   writeArray axs' (m + i) x'
        | i ← [0 .. m − 1]]
      writeArray axs' (m − 1) x
```

First, the number of stored values *n*, the maximum capacity *m*, and the array *axs* are extracted from the references. Since an insertion is going to occur, the reference *pn* is updated to include an extra element. Then, if the number of elements is less than the maximum capacity then the array can be updated at position $m - n - 1$: this is the free slot with the smallest index in the array. Otherwise, a new array *axs'* with maximum capacity $m' = 2 \times m$ is created, and the references *pm* and *paxs* are updated acccordingly. At this point, a sequence of operations are executed, one for each index *i* in the original array, so that the values in *axs* are copied accross to *axs'*. Finally, the new value is written into index $m - 1$, whcih is equal to $m' - n - 1$ when $m = n$, as is true in this case.

Although the *insert* above has amortized constant cost, the return type *ST s* () forces it to only be used inside the context of a *runST* computation. Here is how it might be used to implement an in-place reversal:

```
reverse :: [Int] → [Int]
reverse xs = runST $ do
  pxs ← empty
  sequence [insert x pxs | x ← xs]
  toList pxs
```

This first converst the list into a *ListArray* by inserting the elements one at a time into the array at *pxs*. Then, elements are extracted using *toList*.

18
Bibliography

G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk USSR, Moscow*, 16(2): 263–266, 1962. English translation in Soviet Mathematics, Vol. 3, 1259–1263 (1962).

P. Bachmann. *Die Analytische Zahlentheorie*. Number 2 in Zahlentheorie. Teubner, 1894.

R. Bellman. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957. ISBN 9780691079516.

R. S. Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, New York, NY, USA, 1st edition, 2010. ISBN 0521513383, 9780521513388.

A. Cayley. On the theory of groups, as depending on the symbolic equation $\theta^n = 1$. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 7(42):40–47, 1854. DOI: 10.1080/14786445408647421.

A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

S. Dreyfus. Richard bellman on the birth of dynamic programming. *Operations Research*, 50(1):48–51, 2002.

P. du Bois-Reymond. Sur la grandeur relative des infinis des fonctions. *Annali di Matematica Pura ed Applicata (1867-1897)*, 4(1): 338–353, Jul 1870. ISSN 0373-3114. DOI: 10.1007/BF02420041.

G. H. Hardy. *The integration of functions of a single variable*. Number 2 in Cambridge Tracts in Mathematics and Mathematical Physics. Cambridge University Press, 1905.

G. H. Hardy. Orders of infinity: The 'infinitärcalcül'of Paul du Bois-Reymond. *Cambridge Tracts in Mathematics and Mathematical Physics*, 12, 1910.

G. H. Hardy and J. E. Littlewood. Some problems of diophantine approximation: Part ii. the trigonometrical series associated with the elliptic ϑ-functions. *Acta mathematica*, 37:193–239, 1914.

R. J. M. Hughes. A novel representation of lists and its application to the function "reverse". *Inf. Process. Lett.*, 22(3):141–144, Mar. 1986. ISSN 0020-0190. DOI: 10.1016/0020-0190(86)90059-1.

D. E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, Apr. 1976. ISSN 0163-5700. DOI: 10.1145/1008328.1008329.

E. Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. Teubner, Leipzig, 1909.

P. J. Landin. A correspondence between algol 60 and church's lambda-notations: Part ii. *Commun. ACM*, 8(3):158–167, Mar. 1965. ISSN 0001-0782. DOI: 10.1145/363791.363804.

J. Liouville. Premier mémoire sur la détermination des intégrales dont la valeur est algébrique. *Journal de l'École Polytechnique*, XIV: 124–148, 1833.

D. Michie. "Memo" functions and machine learning. *Nature*, 218 (5136):19, 1968.

D. Sands. Complexity analysis for a lazy higher-order language. In *Functional Programming, Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989, Fraserburgh, Scotland, UK*, pages 56–79, 1989.

D. Sands. *Calculi for time analysis of functional programs*. PhD thesis, Imperial College London, UK, 1990.

B. Schoenmakers. *Data structures and amortized complexity in a functional setting*. PhD thesis, Technische Universiteit Eindhoven, 1992.

P. M. B. Vitányi and L. Meertens. Big omega versus the wild functions. *SIGACT News*, 16(4):56–59, Apr. 1985. ISSN 0163-5700. DOI: 10.1145/382242.382835.

S. von Waltershausen. *Gauss: zum Gedächtnis*. Leipzig : S. Hirzel, 1856.

P. Wadler. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 119–132, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. DOI: 10.1145/73560.73571.