Quicksort

a whistle-stop tour of the algorithm in five languages and four paradigms

Programming Paradigms: Functional, Logic, Imperative, Imperative Functional

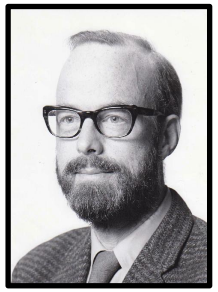
Languages: Haskell, Scala, Java, Clojure, Prolog

Due credit must be paid to the genius of the designers of **ALGOL 60** who included **recursion** in their language and enabled me to **describe my invention** [**Quicksort**] **so elegantly** to the world.

Ask a professional **computer scientist** or **programmer** to list their top 10 algorithms, and you'll find **Quicksort** on many lists, including mine. … On the aesthetic side, **Quicksort is just a remarkably beautiful algorithm**, with an equally beautiful running time analysis.

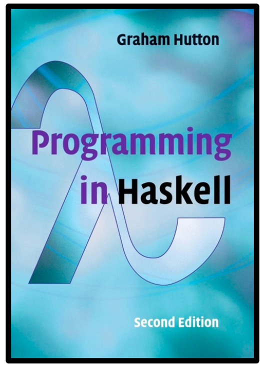
Graham Hutton
@haskellhutt

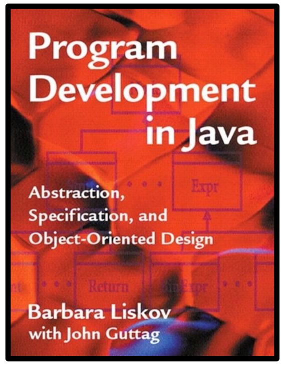
Barbara Liskov

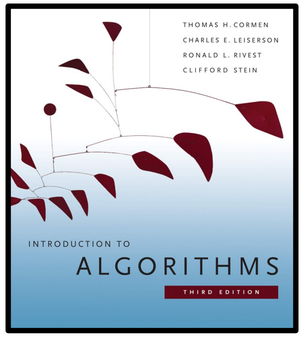

C. Anthony R. Hoare

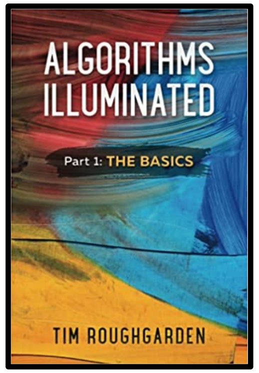
Tim Roughgarden @algo_class

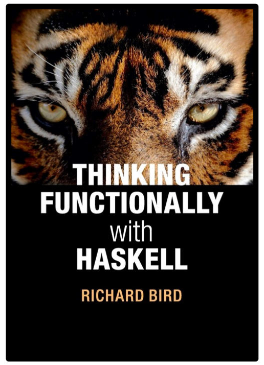
Richard Bird




Program Development in Java
Abstraction, Specification, and Object-Oriented Design
Barbara Liskov with John Guttag

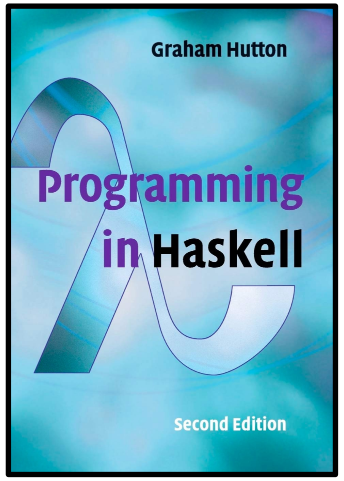






slides by @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

Graham Hutton
@haskellhutt



Sorting values

Now let us consider a more sophisticated function concerning lists, which illustrates a number of other aspects of Haskell. Suppose that we define a function called qsort by the following two equations:

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                 smaller = [a | a <- xs, a <= x]
                 larger  = [b | b <- xs, b > x ]
```

The empty list is already sorted, and any non-empty list can be sorted by placing its head between the two lists that result from sorting those elements of its tail that are smaller and larger than the head

…
This function produces a sorted version of any list of numbers.

The first equation for qsort states that the empty list is already sorted, while the second states that any non-empty list can be sorted by inserting the first number between the two lists that result from sorting the remaining numbers that are smaller and larger than this number.

This method of sorting is called quicksort, and is one of the best such methods known. The above implementation of quicksort is an excellent example of the power of Haskell, being both clear and concise.

Moreover, the function qsort is also more general than might be expected, being applicable not just with numbers, but with any type of ordered values. More precisely, the type qsort :: Ord a => [a] -> [a] states that, for any type a of ordered values, qsort is a function that maps between lists of such values.

Haskell supports many different types of ordered values, including numbers, single characters such as ’a’, and strings of characters such as "abcde".

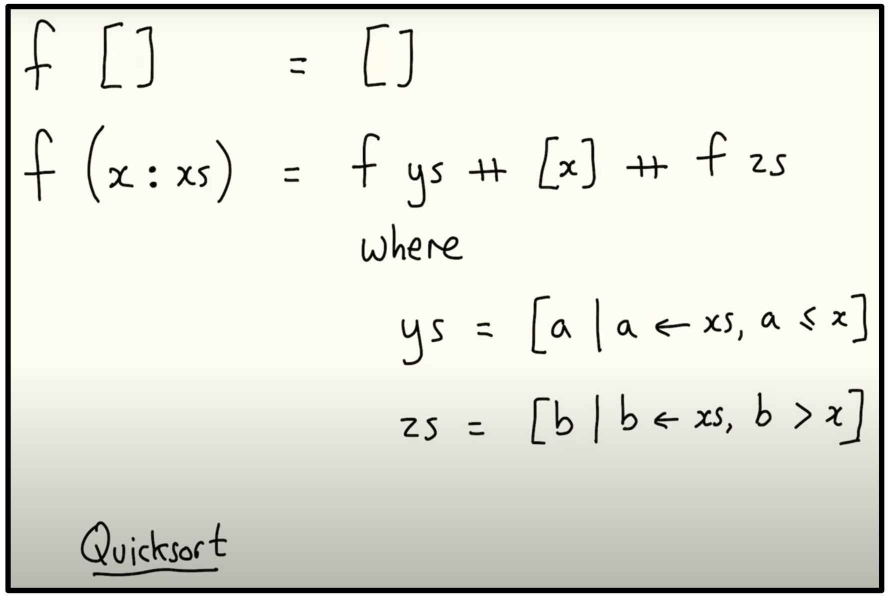
Hence, for example, the function qsort could also be used to sort a list of characters, or a list of strings.

What I wanted to show you today, is what **Haskell** looks like, because it looks quite unlike any other language that you probably have seen.

The program here is very concise, it is only five lines long, there is essentially no junk syntax here, it would be hard to think of eliminating any of the symbols here, and this is in contrast to what you find in most other programming languages.

If you have seen sorting algorithms like Quicksort before, maybe in C, maybe in Java, maybe in an other language, it is probably going to be much longer than this version.

So for me, writing a program like this in **Haskell** catches the **essence** of the **Quicksort** algorithm.



```
f []     = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
             ys = [a | a ← xs, a ≤ x]
             zs = [b | b ← xs, b > x]
```

Quicksort

YouTube Functional Programming in Haskell - FP 3 - Introduction

Graham Hutton
@haskellhutt

The point here at the bottom, it is quite a bold statement, but I actually do believe it is true, **this is probably the simplest implementation of Quicksort in any programming language**.

If you can show me a simpler one than this, I'll be very interested to see it, but I don't really see what you can take out of this program and actually still have the **Quicksort** algorithm.

```
qsort :: Ord a =>  [a] -> [a]
qsort [] = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a <- xs, a <= x]
      larger  = [b | b <- xs, b > x ]
```

This is probably the simplest implementation of **Quicksort** in any programming language!

YouTube Functional Programming in Haskell - FP 8 – Recursive Functions

Here is the Haskell qsort function again, together with the equivalent Scala and Clojure functions.

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                 smaller = [a | a <- xs, a <= x]
                 larger  = [b | b <- xs, b > x ]
```

```scala
def qsort[A:Ordering](as: List[A]): List[A] = as match
  case Nil => Nil
  case x::xs =>
    val smaller = for a <- xs if a <= x yield a
    val larger  = for b <- xs if b > x  yield b
    qsort(smaller) ++ List(x) ++ qsort(larger)
```

```clojure
(defn qsort [elements]
  (if (empty? elements)
     elements
     (let [[x & xs] elements
           smaller (for [a xs :when (<= a x)] a)
           larger  (for [b xs :when (>  b x)] b)]
       (concat (qsort smaller) (list x) (qsort larger)))))
```

And here are some **Haskell** and **Scala** tests for **qsort**.

```
TestCase (assertEqual "sort integers" [1,2,3,3,4,5]               (qsort [5,1,3,2,4,3]))
TestCase (assertEqual "sort doubles"  [1.1,2.3,3.4,3.4,4.5,5.2] (qsort [5.2,1.1,3.4,2.3,4.5,3.4]))
TestCase (assertEqual "sort chars"    "abccde"                    (qsort "acbecd"))
TestCase (assertEqual "sort strings"  ["abc","efg","uvz"]         (qsort ["abc","uvz","efg"]) )
TestCase (assertEqual "sort colours"  [Red,Green,Blue]            (qsort [Blue,Green,Red]))
```

```
data RGB = Red | Green | Blue deriving(Eq,Ord,Show)
```

```
assert(qsort(List(5,1,2,4,3))                        == List(1,2,3,4,5))
assert(qsort(List(5.2,1.1,3.4,2.3,4.5,3.4))          == List(1.1,2.3,3.4,3.4,4.5,5.2))
assert(qsort(List(Blue,Green,Red))                   == List(Red,Green,Blue))
assert(qsort("acbecd".toList)                        == "abccde".toList)
assert(qsort(List ("abc","uvz","efg"))               == List("abc","efg","uvz"))
```

```
enum RGB :
  case Red, Green, Blue
```

```
given Ordering[RGB] with
  def compare(x: RGB, y: RGB): Int =
    x.ordinal compare y.ordinal
```

Let’s use the predefined **filter** function to make the **qsort** functions a little bit more **succinct**.

```haskell
qsort :: Ord a =>  [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                 smaller = [a | a <- xs, a <= x]
                 larger  = [b | b <- xs, b > x ]
```

```haskell
qsort :: Ord a =>  [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                 smaller = filter (x >) xs
                 larger  = filter (x <=) xs
```

```scala
def qsort[A:Ordering](as: List[A]): List[A] = as match
  case Nil => Nil
  case x::xs =>
    val smaller = for a <- xs if a <= x yield a
    val larger  = for b <- xs if b > x  yield b
    qsort(smaller) ++ List(x) ++ qsort(larger)
```

```scala
def qsort[A:Ordering](as: List[A]): List[A] = as match
  case Nil => Nil
  case x::xs =>
    val smaller = xs filter (_ <= x)
    val larger  = xs filter (_ > x)
    qsort(smaller) ++ List(x) ++ qsort(larger)
```

```clojure
(defn qsort [elements]
  (if (empty? elements)
     elements
     (let [[x & xs] elements
           smaller (for [a xs :when (<= a x)] a)
           larger  (for [b xs :when (>  b x)] b)]
       (concat (qsort smaller) (list x) (qsort larger)))))
```

```clojure
(defn qsort [elements]
  (if (empty? elements)
     elements
     (let [[x & xs] elements
           smaller (filter #(<= % x) xs)
           larger  (filter #(>  % x) xs)]
       (concat (qsort smaller) (list x) (qsort larger)))))
```

@philip_schwarz

Now let's improve the qsort functions a bit further by using the predefined partition function.

```haskell
qsort :: Ord a =>  [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                 smaller = filter (x >) xs
                 larger  = filter (x <=) xs
```

```haskell
qsort :: Ord a =>  [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where (smaller,larger) = partition (x >) xs
```

```scala
def qsort[A:Ordering](as: List[A]): List[A] = as match
  case Nil => Nil
  case x::xs =>
    val smaller = xs filter (_ <= x)
    val larger  = xs filter (_ > x)
    qsort(smaller) ++ List(x) ++ qsort(larger)
```

```scala
def qsort[A:Ordering](as: List[A]): List[A] = as match
  case Nil => Nil
  case x::xs =>
    val (smaller,larger) = xs partition (_ <= x)
    qsort(smaller) ++ List(x) ++ qsort(larger)
```

```clojure
(defn qsort [elements]
  (if (empty? elements)
     elements
     (let [[x & xs] elements
           smaller (filter #(<= % x) xs)
           larger  (filter #(>  % x) xs)]
       (concat (qsort smaller) (list x) (qsort larger)))))
```

```clojure
(defn qsort [elements]
  (if (empty? elements)
     elements
     (let [[x & xs] elements
           [smaller larger] (split-with #(<= % x) xs)]
       (concat (qsort smaller) (list x) (qsort larger)))))
```

Like **Haskell**, **Prolog** (the **Logic Programming** language) is also **clear** and **succinct**.

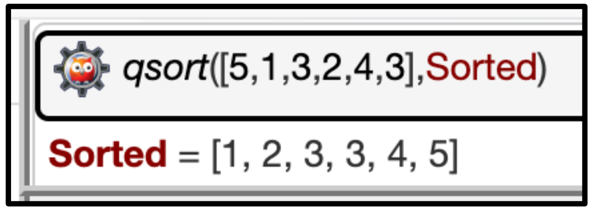
Let’s see how a **Prolog** version of **Quicksort** compares with the **Haskell** one.

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                 smaller = [a | a <- xs, a <= x]
                 larger  = [b | b <- xs, b > x ]
```

```prolog
qsort( [], [] ).
qsort( [X|XS], Sorted ) :-
  partition(X, XS, Smaller, Larger),
  qsort(Smaller, SortedSmaller),
  qsort(Larger, SortedLarger),
  append(SortedSmaller, [X|SortedLarger], Sorted).

partition( _, [], [], [] ).
partition( X, [Y|YS], [Y|Smaller], Larger ) :- Y =< X, partition(X, YS, Smaller, Larger).
partition( X, [Y|YS], Smaller, [Y|Larger] ) :- Y  > X, partition(X, YS, Smaller, Larger).
```

SWI Prolog



```
qsort([5,1,3,2,4,3],Sorted)
Sorted = [1, 2, 3, 3, 4, 5]
```

Here, we first create a variant of the **Scala** **qsort** function that uses **headOption** and **fold**, and then we have a go at writing a somewhat similar **Java** function using **Streams**.

Scala

```scala
def qsort[A:Ordering](as: List[A]): List[A] = as match
  case Nil => Nil
  case x::xs =>
    val smaller = xs filter (_ <= x)
    val larger  = xs filter (_ > x)
    qsort(smaller) ++ List(x) ++ qsort(larger)
```

```scala
def qsort[A:Ordering](xs: List[A]): List[A] =
  xs.headOption.fold(Nil){ x =>
    val smaller = xs.tail filter (_ <= x)
    val larger  = xs.tail filter (_ > x)
    qsort(smaller) ++ List(x) ++ qsort(larger)
  }
```

Java

```java
private static <T extends Comparable> List<T> qsort(final List<T> xs) {
  return xs.stream().findFirst().map((final T x) -> {
    final List<T> smaller = xs.stream().filter(n -> n.compareTo(x) < 0).collect(toList());
    final List<T> larger  = xs.stream().skip(1).filter(n -> n.compareTo(x) >= 0).collect(toList());
    return Stream.of(qsort(smaller), List.of(x), qsort(larger))
                 .flatMap(Collection::stream)
                 .collect(Collectors.toList());
  }).orElseGet(Collections::emptyList);
}
```

Same as the previous slide, but here we use **partition** rather than **filter**.

Scala

```scala
def qsort[A:Ordering](as: List[A]): List[A] = as match
  case Nil => Nil
  case x::xs =>
    val (smaller,larger) = xs partition (_ <= x)
    qsort(smaller) ++ List(x) ++ qsort(larger)
```

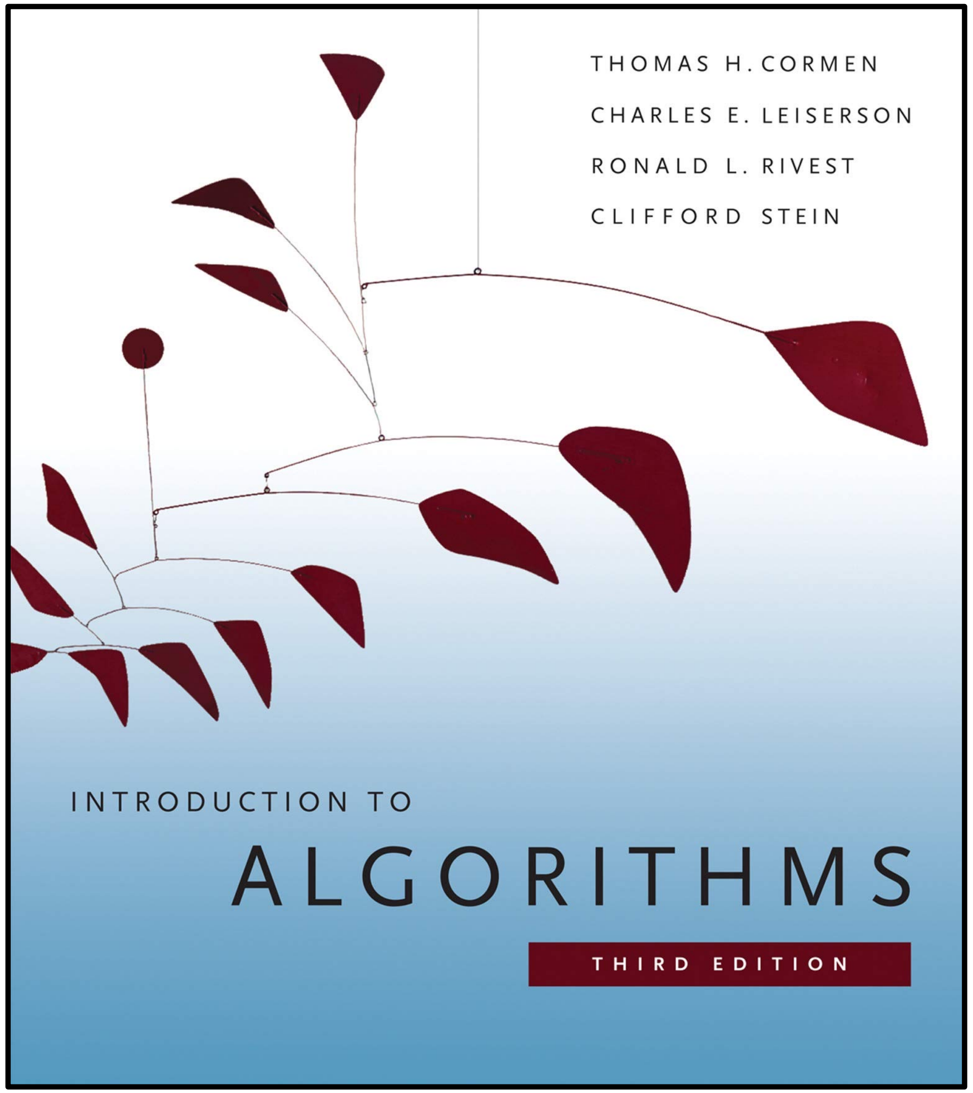
```scala
def qsort[A:Ordering](xs: List[A]): List[A] =
  xs.headOption.fold(Nil){ x =>
    val (smaller,larger) = xs partition (_ <= x)
    qsort(smaller) ++ List(x) ++ qsort(larger)
  }
```

Java

```java
private static <T extends Comparable> List<T> qsort(final List<T> xs) {
  return xs.stream().findFirst().map((final T x) -> {
    Map<Boolean, List<T>> partitions = xs.stream().skip(1).collect(Collectors.partitioningBy(n -> n.compareTo(x) < 0));
    final List<T> smaller = partitions.get(true);
    final List<T> larger  = partitions.get(false);
    return Stream.of(qsort(smaller), List.of(x), qsort(larger))
                 .flatMap(Collection::stream)
                 .collect(Collectors.toList());
  }).orElseGet(Collections::emptyList);
}
```

@philip_schwarz

On the next slide, a brief reminder of the definition of the **Quicksort** algorithm.



Description of quicksort

Quicksort, like **merge sort**, applies the **divide-and-conquer** paradigm. Here is the three-step **divide-and-conquer** process for sorting a typical subarray $A[p..r]$:

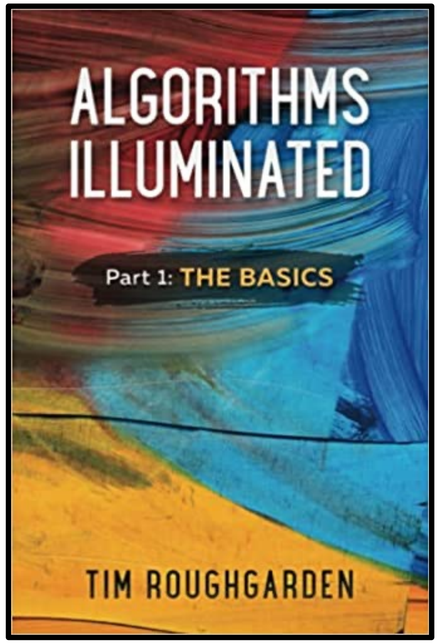
Divide: **Partition** (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this **partitioning procedure**.

Conquer: **Sort** the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by **recursive calls** to **quicksort**.

Combine: Because the subarrays are already **sorted**, no work is needed to **combine** them: the entire array $A[p..r]$ is now **sorted**.

The next slide is a reminder of some of the salient aspects of the **Quicksort** algorithm.

Tim Roughgarden
@algo_class



The Upshot

The famous **Quicksort** algorithm has **three high-level steps**:

1. It chooses one element p of the input array to act as a **pivot element**
2. Its **Partition** subroutine rearranges the array so that elements smaller than and greater than p come before it and after it, respectively
3. It recursively sorts the two subarrays on either side of the **pivot**

The **Partition** subroutine can be implemented to run in **linear time** and **in place**, meaning with negligible additional memory. As a consequence, **Quicksort** also runs **in place**.

The correctness of the **Quicksort** algorithm does not depend on how **pivot elements** are chosen, but its running time does.

The **worst-case scenario** is a **running time** of $\Theta(n^2)$, where n is the length of the input array. This occurs when the input array is already sorted, and the first element is always used as the **pivot element**. The **best-case scenario** is a **running time** of $\Theta(n \log n)$. This occurs when the **median element** is always used as the **pivot**.

In **randomized Quicksort**, the **pivot element** is always chosen **uniformly** at **random**. Its running time can be anywhere from $\Theta(n \log n)$ to $\Theta(n^2)$, depending on its **random** coin flips.

The **average** running time of **randomized Quicksort** is $\Theta(n \log n)$, only a small constant factor worse than its **best-case running time**.

...

A **comparison-based** sorting algorithm is a **general-purpose** algorithm that accesses the input array only by comparing pairs of elements, and never directly uses the value of an element.

No **comparison-based** sorting algorithm has a **worst-case** asymptotic **running time** better than $\Theta(n \log n)$.

ChoosePivot

Input: array A of n distinct integers, left and right endpoints $\ell, r \in \{1,2,\dots,n\}$.
Output: an index $i \in \{\ell, \ell+1, \dots, r\}$.

Implementation:

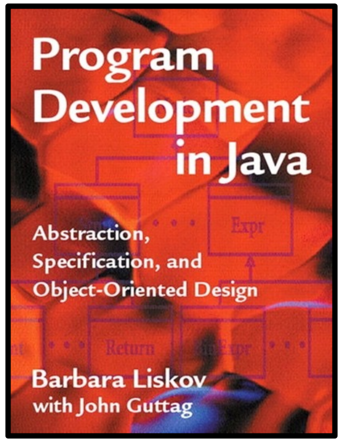
- **Naïve**: return ℓ.
- **Overkill**: return position of the median element of $\{A[\ell], \dots, A[r]\}$.
- **Randomized**: return an element of $\{\ell, \ell+1, \dots, r\}$, chosen uniformly, at random.

On the next slide we see an **imperative** (**Java**) **implementation** of **Quicksort**, i.e. one that does the sorting **in place**.

quick sort … **partitions** the elements of the array into two contiguous groups such that all the elements in the first group are no larger than those in the second group; it continues to **partition recursively** until the entire array is sorted.

To carry out these steps, we use two subsidiary procedures: **quickSort**, which causes the partitioning of smaller and smaller subparts of the array, and **partition**, which performs the **partitioning** of a designated subpart of the array.

Barbara Liskov


Program Development in Java
Abstraction, Specification, and Object-Oriented Design
Barbara Liskov with John Guttag

Note that the **quickSort** and **partition** routines are *not* declared to be public; instead, their use is limited to the **Arrays** class.

This is appropriate because they are just helper routines and have little utility in their own right.

Nevertheless, we have provided specifications for them; these specifications are of interest to someone interested in understanding how **quickSort** is implemented but not to a user of **quickSort**

```
public class Arrays {
    // OVERVIEW: …

    public static void sort (int[ ] a) {
        // MODIFIES: a
        // EFFECTS: Sorts a[0], …, a[a.length - 1] into ascending order.
        if (a == null) return;
        quickSort(a, 0, a.length-1); }

    private static void quickSort(int[ ] a, int low, int high) {
        // REQUIRES: a is not null and 0 <= low & high > a.length
        // MODIFIES: a
        // EFFECTS: Sorts a[low], a[low+1], …, a[high] into ascending order.
        if (low >= high) return;
        int mid = partition(a, low, high);
        quickSort(a, low, mid);
        quickSort(a, mid + 1, high); }

    private static int partition(int[ ] a, int i, int j) {
        // REQUIRES: a is not null and 0 <= i < j < a.length
        // MODIFIES: a
        // EFFECTS: Reorders the elements in a into two contiguous groups,
        //   a[i],…, a[res] and a[res+1],…, a[j], such that each
        //   element in the second group is at least as large as each
        //   element of the first group. Returns res.
        int x = a[i];
        while (true) {
            while (a[j] > x) j--;
            while (a[i] < x) i++;
            if (i < j) { // need to swap
                int temp = a[i]; a[i] = a[j]; a[j] = temp;
                j--; i++; }
            else return j; }
    }
}
```

Yes, that was **Barbara Liskov**, of **Liskov Substitution Principle** fame.

Now that we have seen both **functional** and **imperative** implementations of **Quicksort**, we go back to the **Haskell** **functional implementation** and learn about the following:

- Shortcomings of the **functional implementation**
- How the **functional implementation** can be made more efficient in its use of space.
- How the **functional implementation** can be rewritten in a hybrid **imperative functional** style.

Because the last of the above topics involves fairly advanced **functional programming** techniques, we are simply going to have a quick, high-level look at it, to whet our appetite and motivate us to find out more.

Our second sorting algorithm is a famous one called **Quicksort**. It can be expressed in just two lines of **Haskell**:

```
sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = sort [y | y <- xs, y < x] ++ [x] ++ sort [y | y <- xs, x <= y]
```

That's very pretty and a testament to the expressive power of Haskell. But the prettiness comes at a cost: the program can be very inefficient in its use of space.

Before plunging into ways the code can be optimized, let's compute $T(sort)$. Suppose we want to sort a list of length $n+1$. The first list comprehension can return a list of any length k from 0 to n. The length of the result of the second list comprehension is therefore $n-k$. Since our timing function is an estimate of the **worst-case running time**, we have to take the maximum of these possibilities:

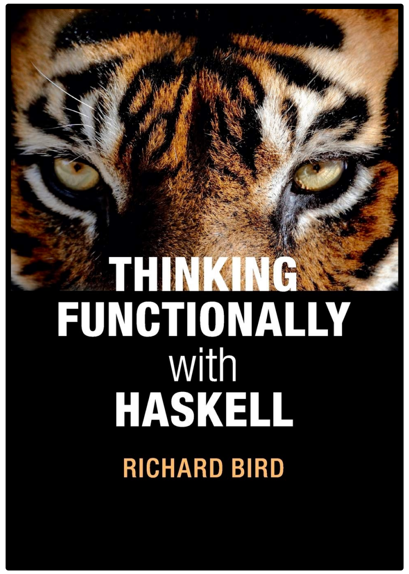
$$T(sort)(n+1) = max[T(sort)(k) + T(sort)(n-k) \mid k \leftarrow [0..n]] + \theta(n).$$

The $\theta(n)$ term accounts for both the time to evaluate the two list comprehensions and the time to perform the concatenation. Note, by the way, the use of a list comprehension in a mathematical expression rather than a **Haskell** one. If list comprehensions are useful notions in programming, they are useful in mathematics too.

Although not immediately obvious, the worst case occurs when $k=0$ or $k=n$. Hence

$$T(sort)(0) = \theta(1)$$
$$T(sort)(n+1) = T(sort)(n) + \theta(n)$$

With solution $T(sort)(n) = \theta(n^2)$. **Thus Quicksort is a quadratic algorithm in the worst case. This fact is intrinsic to the algorithm and has nothing to do with the Haskell expression of it.**



Richard Bird

Quicksort achieved its fame for two other reasons, neither of which hold in a purely functional setting.

Firstly, when Quicksort is implemented in terms of arrays rather than lists, the partitioning phase can be performed in place without using any additional space.

Secondly, the average case performance of Quicksort, under reasonable assumptions about the input, is $\theta(n \log n)$ with a smallish constant of proportionality. In a functional setting this constant is not so small and there are better ways to sort than Quicksort.

With this warning, let us now see what we can do to optimize the algorithm without changing it in any essential way (i.e. to a completely different sorting algorithm).

To avoid the two traversals of the list in the partitioning process, define

```
partition p xs = (filter p xs, filter (not . p) xs)
```

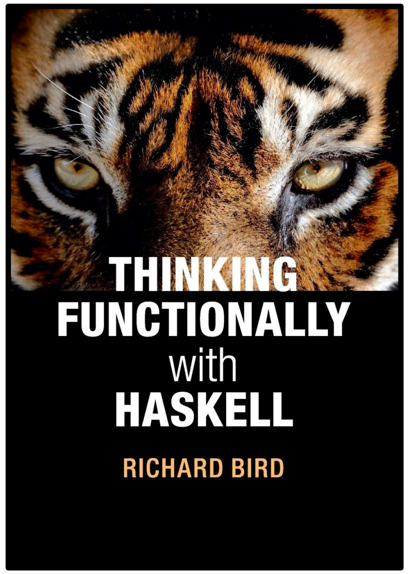
This is another example of **tupling** two definitions to save on a **traversal**. Since `filter p` can be expressed as an instance of **`foldr`** we can appeal to the tupling law of **`foldr`** to arrive at

```
partition p = foldr op ([],[])
              where op x (ys,zs) | p x       = (x:ys,zs)
                                 | otherwise = (ys,x:zs))
```

Now we can write

```
sort []     = []
sort (x:xs) = sort ys ++ [x] ++ sort zs
              where (ys,zs) = partition (< x) xs
```

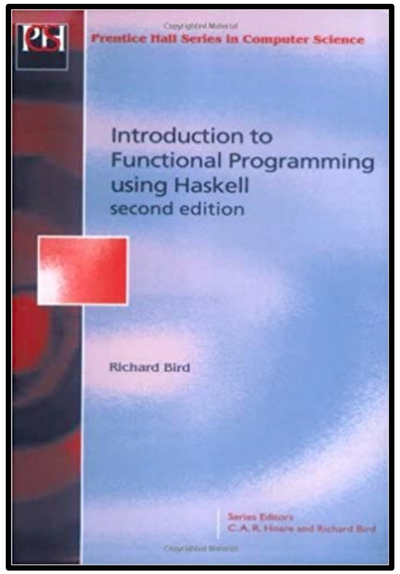
But this program still contains a **space leak**.



Richard Bird

In his earlier book, **Richard Bird** elaborates a bit on **partition**'s **space leak**.

@philip_schwarz


Prentice Hall Series in Computer Science
Introduction to Functional Programming using Haskell second edition
Richard Bird
Series Editors C.A.R. Hoare and Richard Bird

```
partition p = foldr f ([],[])
              where f y (ys,zs) = if p y then (y:ys,zs) else (ys,y:zs))
```

Having rewritten the definition of **partition**, **it may be thought that the program for quicksort is now as space efficient as possible, but unfortunately it is not**.

For some inputs of length n **the space required by the program is** $\Omega(n^2)$.

This situation is referred to as a **space leak**. **A space leak is a hidden loss of space efficiency**. **The space leak in the above program for quicksort occurs with an input which is already sorted, but in decreasing order**.

Richard Bird

To see why, let us write the recursive case in the equivalent form

```
sort (x:xs) = sort (fst p) ++ [x] ++ sort (snd p)
              where p = partition (< x) xs
```

Suppose `x:xs` has length $n + 1$ and is in strictly decreasing order, so `x` is the largest element in the list and `p` is a pair of lists of length n and 0, respectively. Evaluation of `p` is triggered by displaying the results of the first recursive call, but the n units of space occupied by the first component of `p` cannot be reclaimed because there is another reference to `p` in the second recursive call.

Between these two calls further pairs of lists are generated and retained. All in all, the total space required to evaluate sort on a strictly decreasing list of length $n + 1$ is $\theta(n^2)$ units. In practice this means the evaluation of sort on some large inputs can abort owing to a lack of sufficient space.

The solution is to force evaluation of `partition` and, equally importantly, to bind `ys` and `zs` to the components of the pair, not to `p` itself.

One way of bringing about a happy outcome is to introduce two accumulating parameters. Define `sortp` by

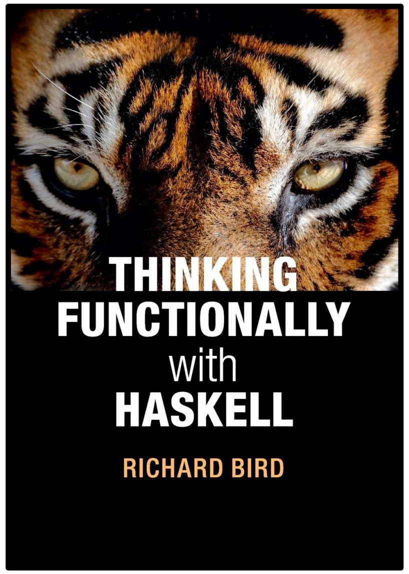
```
sortp x xs us vs = sort (us ++ ys) ++ [x] ++ sort (vs ++ zs)
                   where (ys,zs) = partition (< x) xs
```

Then we have

```
sort (x:xs) = sortp x xs [] []
```

We now synthesise a direct recursive definition of sortp. The base case is

```
sortp x [] us vs = sort us ++ [x] ++ sort vs
```



Richard Bird

For the recursive case `y:xs` let us assume that `y < x`. Then

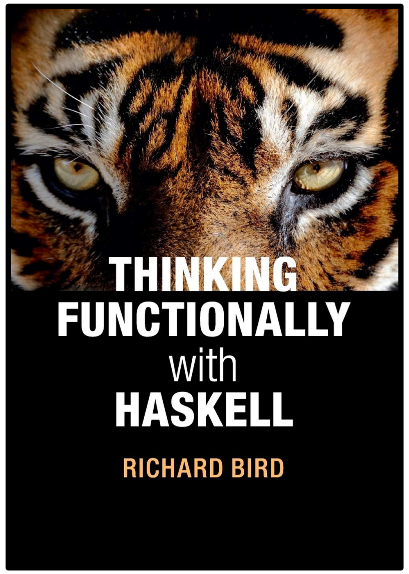
```
  sortp x (y:xs) us vs
=  { definition of sortp with (ys,zs) = partition (<x) xs }
  sort (us ++ y:ys) ++ [x] ++ sort (vs ++ zs)
=  { claim (see below) }
  sort (y:us ++ ys) ++ [x] ++ sort (vs ++ zs)
=  { definition of sortp }
  sortp x xs (y:us) vs
```

The claim is that if `as` is any permutation of `bs` then `sort as` and `sort bs` returns the same result. The claim is intuitively obvious: sorting a list depends only on the elements in the input not their order. A formal proof is omitted.

Carrying out a similar calculation in the case that `x <= y` and making `sortp` local to the definition of `sort`, we arrive at the final program

```
  sort []     = []
  sort (x:xs) = sortp xs [] []
    where sortp [] us vs     = sort us ++ [x] ++ sort vs
          sortp (y:xs) us vs = if y < x
                               then sortp xs (y:us) vs
                               else sortp xs us (y:vs)
```

Not quite as pretty as before, but at least the result has $\theta(n)$ space complexity.



Richard Bird

We have just seen **Richard Bird** improve the **Haskell** **Quicksort** program so that rather than requiring $\Omega(n^2)$ space for some inputs, it now has a space complexity of $\theta(n)$.

$f = O(g)$	f **is of order *at most*** g
$f = \Omega(g)$	f **is of order *at least*** g
$f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$	f **is of order *exactly*** g

Next, **Richard Bird** goes further and rewrites the **Quicksort** program so that it sorts arrays **in place**, i.e. **without using any additional space**.

To do this, he relies on advanced **functional programming** concepts like the **state monad** and the **state thread monad**.

If you are not so familiar with **Haskell** or **functional programming**, you might want to skip the next slide, and skim read the one after that.

Here is the beginning of **Richard Bird**'s explanation of how the **state-thread monad** can be used to model **program variables**.

Imperative Functional Programming

10.4 The ST monad

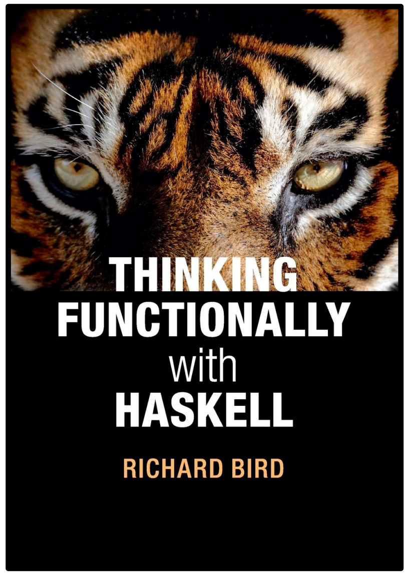
The state-thread monad, which resides in the library `Control.Monad.ST`, is **a different kettle of fish entirely from the state monad, although the kettle itself looks rather similar**. Like `State s a`, you can think of this **monad** as the type

```
  type ST s a = s -> (a,s)
```

but with one very important difference: the **type variable** a cannot be instantiated to specific states, such as `Seed` or `[Int]`. Instead it is there only to ***name*** the state. Think of s as a label that identifies one particular **state *thread***. All mutable types are tagged with this **thread**, so that actions can only affect mutable values in their own **state thread**.

One kind of mutable value is a ***program variable***. **Unlike variables in Haskell, or mathematics for that matter, program variables in imperative languages can change their values**. They can be thought of as ***references*** to other variables, and in **Haskell** they are entities of type `STRef s a`. The s means that the reference is local to the **state thread** s (and no other), and the a is the type of value being referenced. There are operations, defined in `Data.STRef`, to create, read from and write to references:

```
  newSTRef    ::  a -> ST s (STRef s a)
  readSTRef   ::  STRef s a -> ST s a
  writeSTRef  ::  STRef s a -> a -> ST s ()
```



Richard Bird

Here is How **Richard Bird** prepares to rewrite the **Quicksort** program using the **Haskell** equivalent of a **mutable array**.

Imperative Functional Programming

10.5 Mutable Arrays

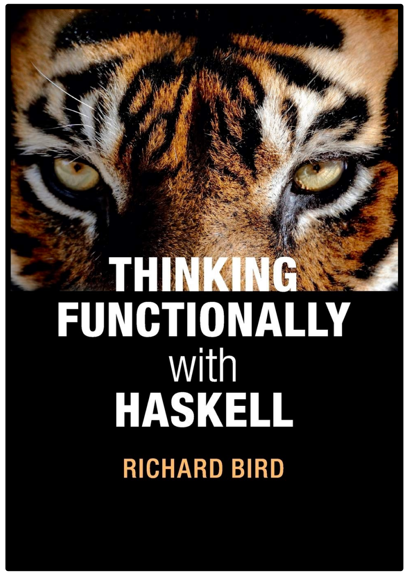
It sometimes surprises **imperative programmers** who meet **functional programming** for the first time that the emphasis is on **lists as the fundamental data structure rather than arrays**. The reason is that most uses of arrays (though not all) depend for their efficiency on the fact that updates are **destructive**. Once you update the value of an array at a particular index the old array is lost. But in **functional programming**, data structures are **persistent** and any named structure continues to exist. For instance, `insert x t` may insert a new element `x` into a tree `t`, but `t` continues to refer to the original tree, so it had better not be overwritten.

In **Haskell** a mutable array is an entity of type `STArray s i e`. The `s` names the **state thread**, `i` the index type and `e` the element type. Not every type can be an index; legitimate indices are members of the type `Ix`. Instances of this class include `Int` and `Char`, things that can be mapped into a contiguous range of integers.

Like `STRefs` there are operations to create, read from and write to arrays. Without more ado, we consider an example explaining the actions as we go along. Recall the **Quicksort** algorithm from Section 7.7:

```
sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = sort [y | y <- xs, y < x] ++ [x] ++
              sort [y | y <- xs, x <= y]
```

There we said that when **Quicksort** is implemented in terms of arrays rather than lists, the partitioning phase can be performed **in place** without using any additional space. We now have the tools to write just such an algorithm.



Richard Bird

@philip_schwarz

Let’s skip the actual rewriting.

The next slide shows the rewritten **Quicksort** program next to the earlier **Java** program.

The **Java** program looks simpler, not just because it is not **polymorphic** (it only handles integers).

```haskell
qsort :: Ord a => [a] -> [a]
qsort xs = runST $
           do {xa <- newListArray (0,n-1) xs;
               qsortST xa (0,n);
               getElems xa}
           where n = length xs
```

```haskell
qsortST :: Ord a => STArray s Int a ->
           (Int,Int) -> ST s ()
qsortST xa (a,b)
  | a == b    = return ()
  | otherwise = do {m <- partition xa (a,b);
                    qsortST xa (a,m);
                    qsortST xa (m+1,b)}
```

```haskell
partition :: Ord a => STArray s Int a ->
             (Int,Int) -> ST s Int
partition xa (a,b)
 = do {x <- readArray xa a;
       let loop (j,k)
           = if j==k
             then do {swap xa a (k-1);
                      return (k-1)}
             else do {y <- readArray xa j;
                      if y < x then loop (j+1,k)
             else do {swap xa j (k-1);
                      loop (j,k-1)}}
       in loop (a+1,b)}
```

```haskell
swap :: STArray s Int a -> Int -> Int -> ST s ()
swap xa i j =  do {v <- readArray xa i;
                   w <- readArray xa j;
                   writeArray xa i w;
                   writeArray xa j v}
```

Haskell

```java
public class Arrays {
    // OVERVIEW: …

    public static void sort (int[ ] a) {
        // MODIFIES: a
        // EFFECTS: Sorts a[0], …, a[a.length - 1] into ascending order.
        if (a == null) return;
        quickSort(a, 0, a.length-1); }

    private static void quickSort(int[ ] a, int low, int high) {
        // REQUIRES: a is not null and 0 <= low & high > a.length
        // MODIFIES: a
        // EFFECTS: Sorts a[low], a[low+1], …, a[high] into ascending order.
        if (low >= high) return;
        int mid = partition(a, low, high);
        quickSort(a, low, mid);
        quickSort(a, mid + 1, high); }

    private static int partition(int[ ] a, int i, int j) {
        // REQUIRES: a is not null and 0 <= i < j < a.length
        // MODIFIES: a
        // EFFECTS: Reorders the elements in a into two contiguous groups,
        //   a[i],…, a[res] and a[res+1],…, a[j], such that each
        //   element in the second group is at least as large as each
        //   element of the first group. Returns res.
        int x = a[i];
        while (true) {
            while (a[j] > x) j--;
            while (a[i] < x) i++;
            if (i < j) { // need to swap
                int temp = a[i]; a[i] = a[j]; a[j] = temp;
                j--; i++; }
            else return j; }
    }
}
```

Java

That’s all.

I hope you found this slide deck useful.