Imperative Programming (Parts 1&2): Sheet 3

Joe Pitt-Francis

Most questions by Mike Spivey, adapted by Gavin Lowe

Suitable for around Week 4, Hilary term, 2019

- Any question marked with † is a self-study question and an answer is provided at the end. Unless your tutor says otherwise, you should mark your attempts at these questions, and hand them in with the rest of your answers.
- Questions marked [**Programming**] are specifically designed to be implemented. Unless your tutor says otherwise, provide your tutor with evidence that you have a working implementation.
- In this sheet, loops should be written as `while`-loops, not `for`-loops.
- You should give an invariant for each non-trivial loop you write, together with appropriate justification.

Question 1

Our binary search code is given in Figure 1.

(a) What does this procedure do if the array `a` is not increasing?

(b) What happens if `N = 0`?

(c) Recall that Scala `Int`s can store only numbers in the range $[-2^{31} \,..\, +2^{31})$. Does the code work correctly with very large arrays? If not, suggest a fix.

Question 2

[**Programming**]

(a) Implement a *ternary search* algorithm[1] to find the integer square root. That is, given positive integer `y`, find a non-negative integer `a` such that $\texttt{a}^2 \leq \texttt{y} < (\texttt{a}+1)^2$, by dividing

[1] Note from *Design and analysis of algorithms* tutorials that it's less attractive than binary search. Ternary search requires fewer iterations but not necessarily fewer comparisons and the code is more complicated.

```
/** Find index i s.t. a[0..i) < x <= a[i..N).
  * Pre: a is sorted. */
def search(a: Array[Int], x: Int) : Int = {
  val N = a.size
  // invariant I: a[0..i) < x <= a[j..N) && 0 <= i <= j <= N
  var i = 0; var j = N
  while(i < j){
    val m = (i+j)/2 // i <= m < j
    if(a(m) < x) i = m+1 else j = m
  }
  // I && i = j, so a[0..i) < x <= a[i..N)
  i
}
```

Figure 1: Binary search from `BinarySearch.scala` lecture code.

each potential range into *three* equally sized portions. Use only `Int` operations. You should check that your function gives the same result as the binary search function developed in lectures for $0 \leq y < \sqrt{2^{31}} \approx 46000$.

(b) When the size of a search interval gets small then does the ternary split ever produce an empty interval? If so, is there any potential for the next iteration of the loop to have an empty interval to work with?

(c) (Optional) Still using the `Int` datatype, remove potential for overflow. Check that your function is correct (produces the same result as a binary search) for all y such that $0 \leq y \leq 10^8$. If you are patient then check up to `Int.MaxValue` $= 2^{31} - 1 = 2147483647$.

Question 3

Adam is thinking of a positive integer X that is unknown to Bill, but he provides a function

```
tooBig(y: BigInt) : Boolean
```

that returns `true` if `y` $> X$ and `false` otherwise.

(a) Suppose Adam tells Bill that X is at most 1000. Help Bill to write a program that uses this function to determine the number X in a constant amount of time. What is the maximum number of calls to `tooBig` that the program makes?

(b) Now suppose Adam does not tell Bill anything about the value of X (other than that it is a positive integer). Help Bill to write a program that determines the number X in a time proportional to $\log_2 X$.

(c) Harder: let ε be any positive constant; show how to determine the value of X using at most $(1+\varepsilon)\log_2 X + r$ invocations of `tooBig`, for some constant r (r may depend on ε, but not on X).

Question 4

If `a[0..n)` is in increasing order, we can use binary search to find an index `k` where a new element `y` can be inserted so that `a` remains in increasing order. Write a procedure that sorts the array `a[0..N)` using the algorithm of *insertion sort*: keep an initial part `a[0..n)` in increasing order, and repeatedly find the correct place to insert `a(n)` into this increasing part, until the whole array is increasing. As `N` becomes large, what is the order of growth of the number of comparisons of elements of `a`? How does the overall running time grow as a function of `N`?

Question 5 †

The code developed in the lecture for Quicksort partition (Figure 2) always uses the leftmost element `a(l)` of a sub-array as the pivot. For what inputs might this be a bad idea and what advantage is gained by choosing the pivot at random? If the input were to consist of distinct elements and the pivot were choose at random, what (approximately) is the expected size of the larger segment that results from partitioning and how does this effect Quicksort?

Question 6

```
/** Partition the segment a[l..r)
  * @return k s.t. a[l..k) < a[k..r) and l <= k < r */
def partition(l: Int, r: Int) : Int = {
  val x = a(l) // pivot
  // Invariant   a[l+1..i) < x = a(l) <= a[j..r) && l < i <= j <= r
  //          && a[0..l) = a_0[0..l) && a[r..N) = a_0[r..N)
  //          && a[l..r) is a permutation of a_0[l..r)
  var i = l+1; var j = r
  while(i < j){
    if(a(i) < x) i += 1
    else{ val t = a(i); a(i) = a(j-1); a(j-1) = t; j -= 1 }
  }
  // swap pivot into position
  a(l) = a(i-1); a(i-1) = x
  i-1 // position of the pivot
}
```

Figure 2: The code for partition

In the code for `partition`, shown in Figure 2, an element of `a[l..r)` larger than the pivot may move twice before finding its proper place. Give an example to illustrate this.

Suggest an improved version of `Partition` in which each element moves at most once. State an invariant for each loop carefully.

Question 7

Read Section 8.9, "Tail recursion", of *Programming in Scala*. (The authors may somewhat over-state the advantages of the recursive style; nevertheless the equivalence between tail recursion and iteration is important.) You might also want to read the start of the Wikipedia article "Call stack" (`http://en.wikipedia.org/wiki/Call_stack`).

Recall our code for quicksort:

```
def QSort(l: Int, r: Int) : Unit = {
  if(r-l > 1){ // nothing to do if segment empty or singleton
    val k = partition(l,r)
    QSort(l,k); QSort(k+1,r)
  }
}
```

The Scala compiler will replace the second recursive call, `QSort(k+1,r)`, by a jump back to the start of the function (setting `l` appropriately) because it is tail recursive; however, it can't eliminate the first recursive call, because it is not tail recursive.

(a) Write down code using a `while` loop that has the same effect. Your code will still have the recursive call `QSort(l,k)`, but the recursive call `QSort(k+1,r)` will have been replaced.

(b) Nevertheless, with either your code from part (a) or the optimization made by the Scala compiler, the call stack may reach depth `N` (the size of the array). Explain why this is.

(c) Suggest a change to your program from part (a) that ensures the stack never gets deeper than $\lceil \log_2 N \rceil$.

Question 8

(a) What happens if Quicksort is used to sort an array in which all entries are identical? How is the running time related to the size of the array?

(b) What happens if Quicksort is used to sort an array in which several identical entries appear, scattered throughout the array?

(c) In sorting such an array, profiling reveals that the `if` statement in `partition` executes its `else` clause very much more frequently than its `then` clause; why is this? Could the performance problems be solved by replacing `<` by `<=` in the `if` condition?

(d) Design a new version of `partition` that gives better performance when the array contains many equal entries, by dividing the segment `a[l..r)`, where `l < r`, into three subsegments so that for some value `pivot`, we have `a[l..i) < pivot` and `a[i..j) = pivot` and `a[j..r) > pivot`, with `l <= i < j <= r`. Give your procedure the heading

```
def partition(l: Int, r: Int) : (Int,Int) = ...
```

so that it returns a pair `(i,j)` where `i` and `j` give the boundaries between each segment and the next.[2] During the partitioning process, split the array into four pieces with boundaries `i`, `j`, `k` such that `l <= i < j <= k <= r`. Maintain the invariant that `a[l..i) < pivot`, and `a[i..j) = pivot`, and `a[k..r) > pivot`, whilst the values in `a[j..k)` are unknown. There's no need this time to hold onto the pivot in `a(l)`.

(e) Show how your new version of `partition` can be used in a better version of Quicksort.

Answer to question 5 †

If the array is distinct and monotonic decreasing (or increasing) then the pivot will always be larger (or smaller) than the other elements in the segment. Partitioning will produce one full segment and one empty segment, so the recurrence for Quicksort will be:

$$T(n) = T(n-1) + O(n),$$

and we will see the worst-case quadratic behaviour. Even if the array is nearly sorted, choosing `a(l)` as the pivot will tend to choose a pivot that is smaller than most of the elements in the segment `a[l..r)`, so partitioning will tend to produce one sub-segment that is larger than the other, and quadratic behaviour will result. With random selection, we cannot guarantee that a bad sequence of pivots will not be chosen, for such a sequence exists whatever the input file, but at least this very unlikely eventuality is independent of any order present in the input.

Assuming that the partition splits the array at a uniform random position then the expected size of the larger sub-array is the same fraction of the original as the expected length of a piece of string obtained by cutting piece of unit length at a random point $x \in [0, 1]$. By symmetry, we may assume that $x \geq 0.5$, so the expected length is $3/4$. The long-term behaviour of Quicksort is then governed by the approximate recurrence

$$T(n) = T(\lfloor 3n/4 \rfloor) + T(\lfloor n/4 \rfloor) + O(n).$$

Even though this recurrence relation is not balanced, its solution is still $O(n \log n)$.

[2]The type `(Int,Int)` represents a pair of `Int`s, just like in Haskell; it is sometimes written as `Tuple2[Int,Int]`. Pairs in Scala are first-class values, and are treated much like in Haskell: you can use commands such as `return (i,j)`, or `val (i,j) = partition(l,r)`. You might want to read "Step 9. Use tuples" in Chapter 3 of *Programming in Scala*.