IP Lecture 8: Quicksort

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

Quicksort in a Haskell style

Recall the quicksort algorithm in Haskell

```
qSort :: [Int] -> [Int]
qSort [] = []
qSort [x] = [x] -- optional
qSort (x:xs) = qSort(filter (<x) xs) ++ [x] ++ qSort(filter (>=x) xs)
```

Scala has a type `List[A]` corresponding to the Haskell type `[A]` of lists containing elements of type `A` (more details later). Here's a version of quicksort following the Haskell definition.

```
def qSort(l: List[Int]) : List[Int] = {
 l match {
  case Nil => l    // l is empty
  case x::Nil => l // l is singleton - optional
  case x::xs  => qSort(xs.filter(_<x)) ++ List(x) ++ qSort(xs.filter(_>=x))
 }
}
```

Profiling

On my machine this takes about 2800ms to sort a list of size 1,000,000.

This is pretty good, but it doesn't seem to scale very well:

List size	Time (seconds)
500,000	1.2
1,000,000	2.8
1,500,000	4.9
2,000,000	18.9
2,500,000	[JVM out of memory]

Profiling suggests that about half the time is spent on the filtering, and about 40% of the time is spent on the appending.

Profiler

A quick-and-dirty Profiler is to wrap bits of code up into functions which start/stop a clock:

```
/** The main quicksort algorithm with profiling*/
def qSort(l: List[Int]) : List[Int] = {
 l match {
  case Nil => l;   case x::Nil => l // l is empty or singleton
  case x::xs  => {
    var ys, zs = List[Int]()
    // The following code is fed to a function, which adds timing.
    Profiler.time("filter"){
      ys = xs.filter(_ < x); zs = xs.filter(_ >= x)
    }
    val ys1 = qSort(ys); val zs1 = qSort(zs)
    // The following code is fed to a function, which adds timing.
    Profiler.time("append"){ ys1 ++ List(x) ++ zs1 }
  }
}
```

Outline of Profiler's time function

```
/** Time cmd, recording the duration against timer tname. */
 def time[A](tname:String)(cmd: => A) : A = {
   // Find the index for the timer tname
   var i = 0
   while(i<tCount && tnames(i)!=tname) i += 1
   if(i==tCount){ tnames(i) = tname; tCount += 1 }
   // start the timer
   val t0 = java.lang.System.currentTimeMillis()  // START CLOCK
   // run the code
   try{
     cmd    // <--- Here's the code
   } finally {
      // STOP CLOCK
     val duration = java.lang.System.currentTimeMillis()-t0
     times(i) += duration // RECORD IT
   }
 }
```

In-situ sorting

The reason the previous version was comparatively slow is that a lot of time is spent creating new lists and copying data between lists. It is far better to use an array, and do all the sorting within that array.

To do this, the recursive calls will correspond to sorting a segment of the array. We'll therefore define the main `QSort` function as follows

```
/** Sort the segment a[l..r), in situ
  * post: $a[0..l) = a_0[0..l) \land a[r..N) = a_0[r..N) \land$
  *       $a[l..r)$ is a sorted permutation of $a_0[l..r)$ */
def QSort(l: Int, r: Int) : Unit = ...
```

and `a` will be a global variable.

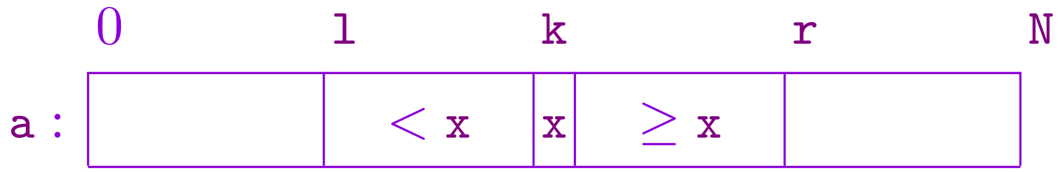
Conventionally, in post-conditions, `a`$_0$ represents the value of `a` at the start of the procedure call.

In-situ quicksort

We will need a function

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

that partitions the non-empty segment `a[l..r)`; i.e., if the value of `a[l..r)` is initially of the form `x:xs`, then it will rearrange these elements into the order `filter (<x) xs ++ [x] ++ filter (>=x) xs` (and leave the rest of `a` alone), and also return the index `k` where `x` ends up.



$$
\begin{aligned}
\textbf{post:}\ & a[0..l) = a_0[0..l) \land a[r..N) = a_0[r..N) \land \\
& a[l..r) \text{ is a permutation of } a_0[l..r) \land \\
& \textbf{returns}\ \ k \textbf{ s.t. } l \le k < r \land a[l..k) < a(k) \le a[k+1..r)
\end{aligned}
$$

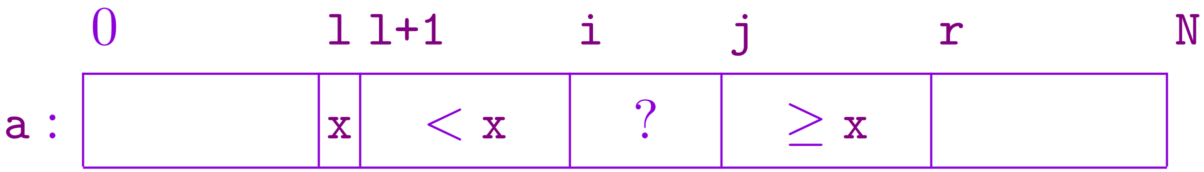
In-situ quicksort

To sort the segment `a[l..r)`, we do this partition, then recursively sort the segments `a[l..k)` and `a[k+1..r)`.

```
/** Sort the segment a[l..r), in situ */
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)
  }
```

Partitioning

To partition the segment `a[l..r)`, we initially keep `x` (the pivot) in position `l`, and rearrange the other elements following this pattern:



Or in symbols

$$\mathtt{a}[\mathtt{l}+1..\mathtt{i}) < \mathtt{x} = \mathtt{a}(\mathtt{l}) \leq \mathtt{a}[\mathtt{j}..\mathtt{r}) \wedge \mathtt{l} < \mathtt{i} \leq \mathtt{j} \leq \mathtt{r}$$

and in addition

$$\mathtt{a}[0..\mathtt{l}) = \mathtt{a}_0[0..\mathtt{l}) \wedge \mathtt{a}[\mathtt{r}..\mathtt{N}) = \mathtt{a}_0[\mathtt{r}..\mathtt{N}) \wedge \mathtt{a[l..r)} \text{ is a permutation of } \mathtt{a}_0\mathtt{[l..r)}$$

When `i=j`, we swap `x` with the element in position `i-1`, and return `i-1`.

Partitioning

```
/** 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
}
```

Profiling

This version takes about 75ms for an array of size 1,000,000 (values randomly chosen in the range `[0..1,000,000)`), compared with about 2800ms for the earlier version.

However, if we test the program with the array entries taking values in the range `[0..100)`, it takes about eight seconds. Profiling in this case shows that the "else" branch is executed many more times than the "then" branch. Exercise: what's going on?

List size	Time (sec)	Time (sec) [< 100]
500,000	0.035	0.890
1,000,000	0.075	3.525
1,500,000	0.110	7.900
2,000,000	0.155	14.020
2,500,000	0.200	22.010

Summary

- Quicksort;
- in-situ partitioning;
- informal profiling.
- Next time: Maximum segment sums & finish Part 1.