DTU Compute
Michael R. Hansen

11/09 2019

02157 Functional Programming
Merge Sort: Exercises in connection with Lecture 2

You shall develop a simple version of *merge sort*, an interesting sorting algorithm that has an $n \log n$ worst-case execution time. The purpose of this particular exercise is to get practice in the development of elegant functional programs on lists – not to develop efficient sorting programs. Furthermore, you should achieve a basic understanding of computations of recursive functions on lists.

Strive for succinctness and elegance when you solve this problem — it is important that your programs and program designs can be communicated to other people.

Remarks:

- We shall later in the course study techniques addressing efficiency.
- There are efficient sorting functions in the .NET libraries, for example, `List.sort`.

The *merge sort* algorithm can be expressed by a functional composition using two functions: `merge` and `split`, where `merge` combines two sorted lists into a single sorted list, and `split` extracts to lists of almost the same sizes from a given list.

The `merge` function

A *merge* of two sorted lists, e.g. merge [1;4;9;12] [2;3;4;5;10;13]) is a new sorted list, [1;2;3;4;4;5;9;10;12;13], made up from the elements of the arguments.

Declare and test this function so that you are sure that all branches of the declaration work correctly.

The `split` function

Declare a function to *split* a list into two lists of (almost) the same lengths. You may declare the function `split` such that

$$\texttt{split}\,[x_0; x_1; x_2; x_3; \ldots; x_{n-1}] \;=\; ([x_0; x_2; \ldots], [x_1; x_3; \ldots])$$

Declare and test this function so that you are sure that all branches of the declaration work correctly.

The `sort` function

The idea behind *top-down* merge sort is a recursive algorithm: take an arbitrary list xs with more than one element and split it into two (almost) equal-length lists: xs_1 and xs_2. Sort xs_1 and xs_2 and merge the results. The empty list and lists with just one element are the base cases.

Declare a function for top-down merge sort in F#.

Test this function so that you are sure that all branches of the declaration work correctly.

Simple performance tests

The function `randomList` *n range* (see `www.imm.dtu.dk/courses/02157/plan.html`) generates a random list of length n containing integers between 0 and *range*:

```
let randomList n range = let rand = let gen = new System.Random()
                                    (fun max -> gen.Next(max))
                         List.init n (fun _ -> rand range);;
```

By use of the toggle `#time;;` (see Page 203 in the textbook) you can measure the computation time of expressions in the interactive F# environment. It takes, for example, 7 milliseconds (of *real time*) to sort a list with 30.000 elements using `List.sort`.

```
> let xs30000 = randomList 30000 1000000;;
val xs30000 : int list =
  [601720; 623254; 419482; 809447; 974642; 338449; 36665; 883508; ... ]
> #time;;
--> Timing now on

> List.sort xs30000;;
Real: 00:00:00.007, CPU: 00:00:00.015, GC gen0: 0, gen1: 0, gen2: 0
val it : int list =
  [29; 36; 39; 68; 125; 144; 167; 171; 220; 258; 273; 296; 354; ... ]
```

Test the running time of your sorting function on lists of sizes 10, 100, 1000, 10000, ...

Remarks:

- The problems with stack overflow you most likely ran into with list sizes between 10000 and 100000 can easily be eliminated using the techniques from Chapter 9. The compiled code will also be much faster when using these techniques.
- The function `List.sort` is implemented by converting the list to an array and use a highly efficient *in place* array-sorting algorithm. This gives a much better algorithm than the simple one described above (even when the efficiency considerations of Chapter 9 has been taken into account).