Programming Assignment #6 (SML)

CS–671

due 17 Apr 2014 11:59 PM

1 Merge Sort and Quick Sort

Merge Sort and *Quick Sort* are two efficient recursive algorithms for sorting an array (or, in the case of this assignment, a list). They both work by breaking a list into two parts and by recursively sorting both parts. The key difference is:

- *Quick Sort* splits a list in two parts so that all the values in one part are smaller than all the values in the other part. This way, after both parts are recursively sorted, they can simply be concatenated.
- *Merge Sort*, on the other hand, splits a list in two parts that do not necessarily enjoy the previous property. The splitting is easier this way, but the two lists cannot simply be concatenated after they are recursively sorted: they have to be merged.

1. Write a function `mergeSort` that implements the *Merge Sort* algorithm: 10 pts

   ```
   mergeSort: ('a * 'a -> bool) -> 'a list -> 'a list
   ```

   The first parameter is a boolean function used to compare elements from the given list (similar to the first parameter of function `checkSorted` from class). Therefore, `mergeSort(Int.<)` is a function of type `int list -> int list` that sorts a list of integers in increasing order, while `mergeSort(String.>)` is a function of type `string list -> string list` that sorts a list of strings in decreasing order.

2. Write a function `quickSort` that implements the *Quick Sort* algorithm. This function has the same type as `mergeSort`. 10 pts

2 Change Function

We consider the problem of giving change using coins and bills from a cash register. We model coins and bills as integers and the cash register as a list of positive integers (possibly with duplicates). The problem becomes one of finding in a given list a sublist of integers that add up to a given target. The simplest algorithm to solve this problem is the following:

Given a list of integers L and a target t:

i. If the target t is zero, the problem is trivially solved with an empty list.

ii. Otherwise, if the list L is empty, the problem has no solution and the algorithm fails.

iii. Otherwise, pick a number x from L and let L' be the list of remaining numbers from L after x is removed. If x is larger than t, it is certainly unusable (since all numbers are positive). Discard it and try to make t with L' through a recursive call. If x is not larger than t, we can try to use it. So, try to make $t - x$ with L' using a recursive call.

   - If this succeeds, we have a list of integers from L' that add up to $t - x$. Add x to the list to get a list of integers from L that add up to x, which is a solution to the original problem.
   - Otherwise, if the algorithm fails to make $t - x$ with L', it means that x shouldn't be used. Discard it and try to make t with L' using a recursive call.

This algorithm uses a programming technique known as *backtracking*: try something and, if it doesn't work, backtrack and try something else. In this case, the choice is between two branches: to use x or not. In order to implement backtracking, we need a way for the program to fail and to continue after failure. One way to implement a failure is to use exceptions; another way is to use SML polymorphic `option` type and use `NONE` to represent failure.

3. Write a function `change` of type `int list -> int -> int list` that implements the previous algorithm. The function raises an exception `CannotChange` if the problem has no solution. For instance: 25 pts

   `change [4,2,32,5,7,1,33,4,1,6] 41` may return `[4,2,1,33,1]`

   (or some other numbers from the list that add up to 41), while

   `change [4,2,32,5,7,1,33,4,1,6] 64` should raise the exception `CannotChange`

   Backtracking can be implemented using the exception `CannotChange` directly or using options. In the latter case, the function must still raise `CannotChange` for unsolvable problems.

4. Write a function `changeBest` of type `int list -> int -> int list` that solves the change problem with a list of minimal length. To achieve this, the algorithm described earlier needs to be modified slightly: the function explores the second branch of the search even when the first branch results in a solution. If both branches lead to solutions, it picks the shortest. If there is no solution at all, the function raises `CannotChange`. 10 pts

   Note that there is no way to know a priori which branch will result in the shortest solution, no matter how cleverly sorted the list of numbers might be. The function has to explore both branches to guarantee a solution of minimal length.

3 Su Doku

In this last part of the assignment, we want to apply the same backtracking approach to solve *Su Doku* puzzles. A *Su Doku* puzzle consists of a 9×9 grid partially filled with numbers.

		8	1			7		
2	4	7			8			
			6			3		8
					2	5		
				3				
		2	7					
9		6			7			
			9			2	1	6
		5			1	4		

The objective of the puzzle is to complete the grid with digits 1 through 9 so that:

i. each line contains all 9 digits

ii. each column contains all 9 digits

iii. each major 3×3 square contains all 9 digits

For instance, the previous puzzle can be solved as follows:

6	3	8	1	2	5	7	9	4
2	4	7	3	9	8	6	5	1
1	5	9	6	7	4	3	2	8
8	9	3	4	1	2	5	6	7
4	7	1	5	3	6	9	8	2
5	6	2	7	8	9	1	4	3
9	1	6	2	4	7	8	3	5
7	8	4	9	5	3	2	1	6
3	2	5	8	6	1	4	7	9

The algorithm to be used is similar to the backtracking algorithm of the `change` functions above, except that there can be more than two branches to choose from:

i. If the grid is filled with numbers, the problem is solved.

ii. Otherwise, pick an empty square x and compute all the digits that are possible values for x by looking at x's row, column and 3×3 square (these possible values are the "branches").

iii. If this list is empty, the attempt fails.

iv. Otherwise, assign x with the first possible digit and try to fill the remaining empty slots with a recursive call.

v. If this fails, try the next possible value for x.

vi. If all values fail, the attempt fails.

If the initial grid admits at least one solution, this algorithm will find one. If the initial grid is consistent (the same digit does not appear twice in the same row, column or 3×3 square) but has not solution, this algorithm will fail. If the initial grid is inconsistent, the algorithm may fail or may return an invalid solution.

5. Implement this algorithm as a structure `ListSudoku` of signature `SUDOKU`. 45 pts

This structure implements a type `grid`, two exceptions `Parse` and `Unsolvable`, and the functions below. The type `grid` is implemented as a list of 81 integers from the range 0 through 9. These integers represent the numbers in the grid, from top to bottom and from left to right. Zeros represent empty squares.

- `parseString: string -> grid`
  Parses a string into a grid. It fails with a `Parse` exception if the string cannot be parsed. Spaces and newlines in the string are ignored and any character other than digits 1 through 9 represents an empty space in the grid. Squares are listed from top to bottom and left to right. For instance, the puzzle above can be represented by any one of the following strings:

```
". . 8 1 . . 7 . .
 2 4 7 . . 8 . . .
 . . . 6 . . 3 . 8
 . . . . . 2 5 . .
 . . . . 3 . . . .
 . . 2 7 . . . . .
 9 . 6 . . 7 . . .
 . . . 9 . . 2 1 6
 . . 5 . . 1 4 . ."

"008100700247008000000600308000002500000030000002700000906007000000900216005001400"
```

- `parseFile: string -> grid`
  Parses the content of a file according the the scheme described with the `parseString` function.

- `print: grid -> unit`
  Prints a grid on the terminal as a two dimensional array of digits. Empty squares are represented with dots.

- `get: grid * int -> int`
  `get(g,i)` returns the number at position `i` in grid `g` (including 0 for empty slots).

- `set: grid * int * int -> grid`
  `set(g,i,x)` returns a grid identical to `g` except that the new grid contains `x` at position `i`. In this implementation, the original grid is a list and is not modified.

- `unsolved: grid -> int option`
  Returns the index of the first empty square in the grid, or `NONE` if the grid is full.

- `possibles: grid * int -> int list`
  `possibles(g,i)` returns a list (in any order) of the digits that may be used in position `i`, based on the digits in the other squares (the digit currently at position `i`, if any, is ignored).

- `valid: grid -> bool`
  Returns *true* if and only if a grid is entirely filled and is consistent.

- `sudoku: grid -> grid`
  Applies the algorithm above to the given grid and returns a full grid solution to the puzzle. The grid that is returned is guaranteed to be valid. This function fails with the `Unsolvable` exception if it cannot find a solution (including in the case when the initial grid is inconsistent)

Notes:

- This assignment must be submitted in a file named `6.sml`. This file can load other files using function `use`, if necessary. Do not resubmit `sudoku-sig.sml`.
- The following function returns *true* exactly when the digits at positions `i` and `j` in a *Su Doku* grid must be different (assuming `i` $\neq$ `j`):

```
fun conflict (i,j) =
    i mod 9 = j mod 9 orelse (* same column *)
    i div 9 = j div 9 orelse (* same row *)
    i div 27 = j div 27 andalso i mod 9 div 3 = j mod 9 div 3 (* same 3x3 *)
```

  It can be useful in writing some functions of the structure, in particular `possibles` and `valid`.
- Function `sudoku` can use exceptions or options to implement backtracking.
- Using a list of integers as the data structure for a grid leads to inefficiencies (creating new lists to modify one element, using `@`, etc.). Because lists remain small (the grid is 81 elements long and the lists returned by `possible` have at most 9 elements), these inefficiencies are acceptable. I will discuss in class how more advanced features of SML could be used to improve on that, but lists are fine for this assignment.
- The part "`: SUDOKU`" in the header of structure `ListSudoku` ensures that the functions in the structure have the types specified in the signature. It also hides any function defined in the structure that is not part of the signature. In order to start debugging an incomplete structure (and, in particular, to try helper functions), it can be convenient to temporarily remove the "`: SUDOKU`" part. Make sure to put it back before you submit the assignment.
- Functions that are not part of the structure *must* have the following types:

```
val mergeSort = fn : ('a * 'a -> bool) -> 'a list -> 'a list
val quickSort = fn : ('a * 'a -> bool) -> 'a list -> 'a list
val change = fn : int list -> int -> int list
val changeBest = fn : int list -> int -> int list
```

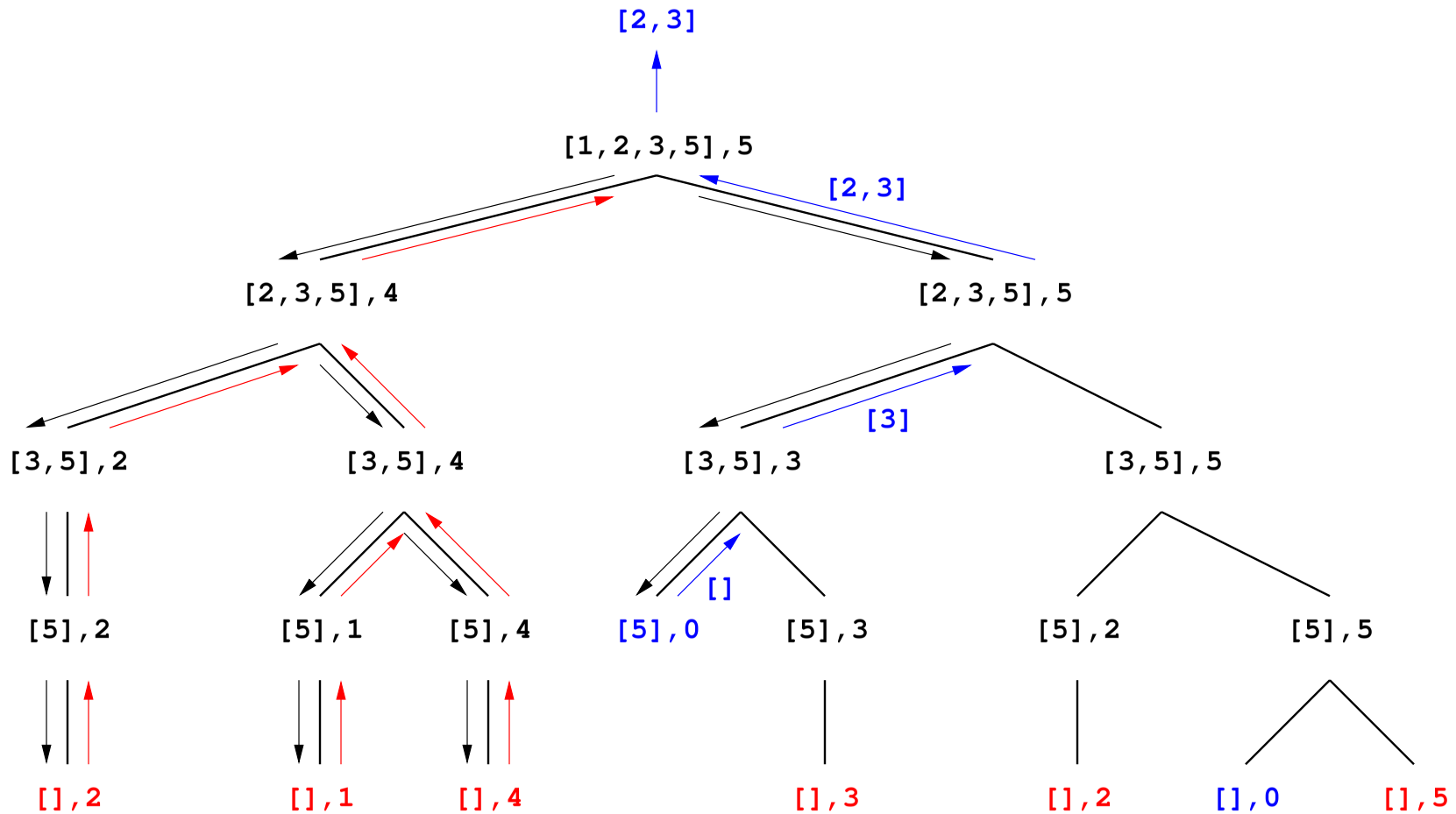
  The structure *must* implement the following signature, which cannot be modified:

```
signature SUDOKU = sig

eqtype grid
exception Parse
exception Unsolvable

val parseString: string -> grid
val parseFile: string -> grid
val get: grid * int -> int
val set: grid * int * int -> grid
val unsolved: grid -> int option
val possibles: grid * int -> int list
val valid: grid -> bool
val print: grid -> unit
val sudoku: grid -> grid
end
```

  This signature and a structure stub have been added to the repositories. Functions `listSudoku.parseFile` and `listSudoku.print` are already implemented and should not need any modification.


[2,3]
[1,2,3,5],5
[2,3]
[2,3,5],4
[2,3,5],5
[3]
[3,5],2
[3,5],4
[3,5],3
[3,5],5
[]
[5],2
[5],1
[5],4
[5],0
[5],3
[5],2
[5],5
[],2
[],1
[],4
[],3
[],2
[],0
[],5