Homework 10: Curried Sorting Algorithms and Vector Operations in Scala

Stephen Wagstaff
CS 431
November 18, 2018

Sorting

1. Write a function `merge_sort(f: (Int, Int) => Boolean)) (lst: List[Int]): List[Int]` that takes a list, and return a sorted list.

```
def merge_sort(f: (Int, Int) => Boolean) (initialList: List[Int]): List[Int] = {
  def split: (List[Int], List[Int]) = {
    val cut = initialList.length/2
    ( initialList take cut , initialList drop cut )
  }
  def merge(leftList: List[Int], rightList: List[Int]): List[Int] = {
    (leftList, rightList) match {
      case (Nil, _) => rightList
      case (_, Nil) => leftList
      case (leftElement::leftRemaining, rightElement::rightRemaining) =>
        if (f(leftElement,rightElement)) leftElement::merge(leftRemaining, rightList)
        else rightElement::merge(leftList, rightRemaining)
    }
  }
  initialList match{
    case Nil => Nil
    case x::Nil => List(x)
    case _ =>
      val (left, right) = split
      merge (merge_sort (f) (left), merge_sort (f) (right))
  }
}
```

2. Write a function `selection_sort(f: (Int, Int) => Boolean)) (lst: List[Int]): List[Int]` that takes a list and return a sorted list.

```
def selection_sort(f: (Int, Int) => Boolean) (initialList: List[Int]): List[Int] = {
    def select(selectList: List[Int]): List[Int] = {
      selectList match {
        case Nil => Nil
        case lastElement::Nil => List(lastElement)
        case firstElement::list =>
          val testElement::remaining = select(list)
          if(f(testElement, firstElement)) select(testElement::firstElement::remaining)
          else firstElement::testElement::remaining
      }
    }
    initialList match {
      case Nil => Nil
      case _ => select(initialList)
    }
}
```

3. Write a function `insertion_sort(f: (Int, Int) => Boolean)) (lst: List[Int]): List[Int]` that takes a list and return a sorted list.

```
def insertion_sort(f: (Int, Int) => Boolean) (initialList: List[Int]): List[Int] = {
    def insert(element: Int, list: List[Int]): List[Int] ={
      (element, list) match {
```

```
        case (_, Nil) => List(element)
        case (_, head::rest) =>
          if(f(element, head)) element::head::rest
          else head::insert(element, rest)
      }
    }
    def sort(sofar: List[Int], list: List[Int]): List[Int] = {
      list match {
        case Nil => sofar
        case element::rest => insert(element, sort(sofar,rest))
      }
    }
    initialList match {
      case Nil => Nil
      case _ => sort(Nil, initialList)
    }
  }
```

Vector and Matrix Operations

1. Write a function `vectorAdd: (List[Int], List[Int]) => List[Int]` that add two integer vectors of the same size.
   - For example, `vectorAdd (List(1,2,3), List(4,5,6))`should return `List(5, 7, 9)`.

```
def vectorAdd(vector1: List[Int], vector2: List[Int]): List[Int] = {
    (vector1, vector2) match {
      case(Nil, Nil) => Nil
      case(Nil, _) => vector2
      case(_, Nil) => vector1
      case(_, _) => vector1.zip(vector2).map({case(int1, int2)=> int1+int2})
    }
  }
```

2. Write a function `svProduct: (Int, List[Int]) => List[Int]` that multiple an integer with an integer list.

- For example, `svProduct(2, List(1,2,3))` should return `List(2,4,6)`.

```
def svProduct(multInt: Int, intList:List[Int]): List[Int]= {
    (multInt, intList) match{
      case(_, Nil) => Nil
      case(_, _) => intList.map((y: Int) => multInt*y)
    }
  }
```

3. Write a function `vmProduct: (List[Int], List[List[Int]]) => List[Int]` that multiple a row vector of size `n` with a matrix with `n` rows and `m` columns to produce a vector of size `m`.

- For example, `vmProduct(List(1,2,3), List(List(1,1), List(2,1), List(3,1)))` should return `List(14, 6)`. Or,

```
                1    1
[1  2  3]× 2    1 = 1×[1  1] + 2×[2  1] + 3×[3  1] = [1  1] + [4  2] + [9  3]= [14  6]
                3    1
```

  This function uses the functions `svProduct` and `vectorAdd` defined earlier.

```
def vmProduct(vector: List[Int], matrix: List[List[Int]]): List[Int] ={
    vector.zip(matrix)
      .map({case(int:Int, list:List[Int])=>svProduct(int,list)})
      .reduce(vectorAdd)
}
```

4. Write a function `matrixProduct: (List[List[Int]], List[List[Int]]) => List[List[Int]]` that multiple a `m×n` matrix with a `n×k` matrix to obtain a `m×k` matrix.

- For example

```
 1    2    3   1    1
 1    1    1 x 2    1 = v1 = 14 6
               3    1   v2 = 6  3

   where
                    1    1
   v1 = [1  2  3] × 2    1 = [14    6]
                    3    1
   and
                   1    1
   v2= [1  1  1] × 2    1 = [6    3]
                   3    1
   That is,
   matrixProduct(List(List(1,2,3), List(1,1,1)),
   List(List(1,1), List(2,1), List(3,1)))
   = List( List(14, 6), List(6, 3) )
```

This problem will use the function `vmProduct` defined previously.

```
def matrixProduct(matrix1: List[List[Int]], matrix2: List[List[Int]]): List[List[Int]] = {
    matrix1.map({row: List[Int] => vmProduct(row, matrix2)})
 }
```