

# Not The Maximum Segment Sum Problem

## Introduction to Functional Programming

Yu Liu

December 15, 2015

*Not the Maximum Segment Sum* (NMSS for short) problem is an interesting problem from the context of [1]. In this short report I introduce how to compute NMSS problem in **liner time** and also introduce some concepts and skills of functional programming (FP). As a related topic, the *Maximum Segment Sum* problem has been studied by many researchers [2–4].

For NMSS problem, there are several ways to code it (please refer to the section 5), however, the functional way introduced in [1] provides a clean, bug-free and runtime efficient algorithm. For Java programmers, such kind of functional thinking will be helpful when handling complicated and miscellaneous problems.

## 1 Notations

The syntax is mainly based on Haskell but a Java implementation is given. List is denoted as  $[x, y, z]$  and  $++$  means list concatenation. Function application is denoted with a space with its argument without parentheses, i.e.,  $f a$  equals to  $f(a)$ . Functions are curried and bound to left and thus  $f a b$  equals to  $(f a) b$ . Functions can be composed by “.” and it has higher priority than other operators.

## 2 The Definition of Non-Segment

By definition of segment, contiguous subsequence of a list, the non-segment is defined: for a list at shortest with 3 elements, the non-segments are the sequences that are not contiguous subsequences (segments), formally, denoted by a regular expression as follows.

$$F^*T^+F^+T(T+F)^*$$

For example, given a list:

$$[-4, -3, -7, +2, +1, -2, -1, -4]$$

The subsequence  $[-4, -3, -7]$  is a segment, while  $[-4, -7]$  is a not-segment and  $[-4, -3, -7, +1]$  is also a not-segment of the original list. We can use a automaton to recognize this regular expression:

$$dataState = E|S|M|N$$

- State E for  $F^*$  (empty)
- State S for  $F^*T^+$  (suffix)
- State M for  $F^*T^+F^+$  (middle)
- State N for  $F^*T^+F^+T(T+F)^*$

The sum of a non-segment is called *Non-Segment Sum* (NSS).

### 3 Specification

The NMSS is to find a NSS which has the maximum sum.

$$\begin{aligned} mnss &:: [Int] \rightarrow Int \\ mnss &= maximum \cdot map \text{sum} \cdot nonsegs \end{aligned}$$

To define the *nonsegs* function, firstly, we introduce a *markings* function:

$$\begin{aligned} markings &:: [a] \rightarrow [[(a, Bool)]] \\ markings \text{ xs} &= [zip \text{ xs bs} | bs \leftarrow \text{booleans}(\text{length xs})] \\ \text{booleans } 0 &= [[]] \\ \text{booleans } (n+1) &= [b : bs | b \leftarrow [True, False], bs \leftarrow \text{booleans } n] \end{aligned}$$

So that, the *nonsegs* can be defined as

$$\begin{aligned} nonsegs &:: [a] \rightarrow [[a]] \\ nonsegs &= \text{extarct} \cdot \text{filter nonsegs} \cdot markings \\ \text{extarct} &:: [[(a, Bool)]] \rightarrow [[a]] \\ \text{extarct} &= \text{map}(\text{map fst} \cdot \text{filter snd}) \end{aligned}$$

By making use of the automaton, *nonsegs* can also be defined as

$$\text{nonseg} = (==N) \cdot \text{foldl step E} \cdot \text{map snd}$$

Here, the *Step E* means start the automaton from E state, and here are the states transformation functions:

$$\begin{array}{ll} \text{step E False} = E & \text{step M False} = M \\ \text{step E False} = S & \text{step M False} = N \\ \text{step S False} = M & \text{step N False} = N \\ \text{step S False} = S & \text{step N False} = N \end{array}$$

### 4 Derivation

The *mnss* can be redefined as

$$\begin{aligned} mnss &= maximum \cdot map \text{sum} \cdot \text{extract} \cdot \text{filter nonseg} \cdot markings \\ \text{extarct} &= \text{map}(\text{map fst} \cdot \text{filter snd}) \\ \text{nonseg} &= (==N) \cdot \text{foldl step E} \cdot \text{map snd} \end{aligned}$$

the problem is turned to find a way to apply the fusion law of *foldl* to get a better algorithm, that *extract · filter nonseg · markings* can be treated as an instance. So, we can define a *pick* function as follows.

$$\begin{aligned} \text{pick} &:: \text{State} \rightarrow [a] \rightarrow [[a]] \\ \text{pick } q &= \text{extract} \cdot \text{filter}((= q) \cdot \text{foldl } \text{step } E \cdot \text{map } \text{snd}) \cdot \text{markings} \end{aligned}$$

just let  $q = N$ ,  $\text{nonsegs} = \text{pick } N$ . Here clime these equations hold:

$$\begin{aligned} \text{pick } xs &= [[ ]] \\ \text{pick } S [] &= [] \\ \text{pick } S (xs ++ [x]) &= \text{map}(++[x])(\text{pick } S xs ++ \text{pick } E xs) \\ \text{pick } M [] &= [] \\ \text{pick } M (xs ++ [x]) &= \text{pick } M xs ++ \text{pick } S xs \\ \text{pick } N [] &= [[ ]] \\ \text{pick } N (xs ++ [x]) &= \text{pick } N xs ++ \text{map}(++[x])(\text{pick } N xs ++ \text{pick } M xs) \end{aligned}$$

recast the definition of *pick* as an instance of *foldl*: firstly we make a tuple:

$$\text{pickall}xs = (\text{pick } E xs, \text{pick } S xs, \text{pick } M xs, \text{pick } N xs)$$

so, we have

$$\begin{aligned} \text{pickall} &= \text{foldl } \text{step}([[], [], [], []]) \\ \text{step}(ess, nss, mss, sss)x &= (ess, \text{map}(++[x])(sss ++ ess), \\ &\quad mss ++ sss, nss ++ \text{map}(++[x])(nss ++ mss)) \end{aligned}$$

so that  $mnss = \text{maximum} \cdot \text{map } \text{sum} \cdot \text{fourth} \cdot \text{pickall}$ . by define

$$\text{tuple } f(w, x, y, z) = (f w, f x, f y, f z).$$

Then we have :

$$\text{maximum} \cdot \text{map } \text{sum} \cdot \text{fourth} = \text{fourth} \cdot \text{tuple}(\text{maximum} \cdot \text{map } \text{sum})$$

Then  $mnss$  is changed to:  $mnss = \text{fourth} \cdot \text{tuple}(\text{maximum} \cdot \text{map } \text{sum}) \cdot \text{pickall}$

By fusion law of *foldl*:

$$f(\text{foldl } g a xs) = \text{foldl } h b xs$$

if  $f a = b$ , and  $f(g x y) = h(f x)y$  hold for all  $x$  and  $y$ .  
f,g,and  $a$  are instantiations:

$$\begin{aligned} f &= \text{tuple}(\text{maximum} \cdot \text{map } \text{sum}) \\ g &= \text{step} \\ a &= ([[], [], [], []]) \end{aligned}$$

We need to find the  $h$  and  $b$ , satisfy the fusion conditions.

$$\text{tuple}(\text{maximum} \cdot \text{map } \text{sum})([[], [], [], []]) = (0, -\infty, -\infty, -\infty)$$

Here  $h$  and  $b$  is given:  $b$  is  $(0, -\infty, -\infty, -\infty)$  and

$$h(e, s, m, n)x = (e, (s \uparrow e) + x, m \uparrow s, n \uparrow ((n \uparrow m) + x))$$

so that

$$mnss = fourth \cdot foldl \ h(0, -\infty, -\infty, -\infty)$$

to simplify the definition of  $mnss$  we can do this :

$$\begin{aligned} mnss \ xs &= fourth(foldl \ h \ (start(take \ 3 \ xs)) \ (drop \ 3 \ xs)) \\ start \ [x, y, z] &= (0, \uparrow [x + y + z, y + z, z], \uparrow [x, x + y, y], x + z) \end{aligned}$$

## 5 Remarks

For current definition of *not the segment*, there is a trivial solution that one can keep all positive numbers and then check if they are conjunctive (during one-pass scan of the input list). In case all positive numbers are not in a segment, the NMSS can be obtained directly, while if all positive numbers are in a segment, one can either remove a positive or introduce a negative number to make a not-segment and get the sum (one must memorize the minimum 3 positive and maximum 3 negative numbers for a constant-number of cases due to the positions where they appear: on the first or last spot of the sequence). In such a way, the NMSS could be calculated in  $O(n)$  time (with miscellaneous edge-condition-handling). If we restrict the definition of NMSS that we do not allow any conjunctive element from original list, then the problem become more difficult and previous strategy does not work.

Also, for at-least-length- $k$  problem we can derive  $O(nk)$  algorithm. And, for non-regular conditions such as  $F^*T^nF^*T^nF^*$  ( $n \geq 0$ ) is susceptible to the same method. The Java implementation is here: <https://github.com/moyun/Mnss>. Readers can evaluate the performance of the linear algorithm and compare with your own implementation.

## References

- [1] Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
- [2] M.~Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problems. Report {csr}-25-93, Department of Computing Science, The University of Edinburgh, 1993.
- [3] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 146–155. ACM Press, jun 2007.
- [4] Z.~Hu, H.~Iwasaki, and M.~Takeichi. Formal Derivation of Parallel Program for 2-Dimensional Maximum Segment Sum Problem. In *Annual European Conference on Parallel Processing, LNCS 1123*, pages 553–562. Springer-Verlag, 1996.