Challenge 1
Monotonic Segments and GHC Sort

VerifyThis at ETAPS 2019
Organizers: Claire Dross, Carlo A. Furia,
Marieke Huisman, Rosemary Monahan, Peter Müller

6–7 April 2019, Prague, Czech Republic

How to submit solutions: send an email to `verifythis19@googlegroups.com` with your solution in attachment. Remember to clearly identify you, stating your group's name and its members.

This challenge[1] is in two parts, each consisting of several different verification tasks. Since you have limited time at your disposal, you are *not* expected to solve both challenges: pick the one that you find the most feasible given the tool you're using and your preferences.

You can also choose to focus on specific verification tasks and assume the others; for example, you can just assume that a procedure works according to some specification without actually implementing or proving it.

1 Part A: Monotonic Segments

Given a sequence s

$$s \quad = \quad s[0]\, s[1]\, \ldots\, s[n-1] \qquad n \geq 0$$

of elements over a totally sorted domain (for example, the integers), we call **monotonic cutpoints** any indexes that cut s into segments that are monotonic: each segment's elements are all increasing or all decreasing. Here are some examples of sequences with monotonic cutpoints:

SEQUENCE s	MONOTONIC CUTPOINTS	MONOTONIC SEGMENTS
1 2 3 4 5 7	0 6	1 2 3 4 5 7
1 4 7 3 3 5 9	0 3 5 7	1 4 7 | 3 3 | 5 9
6 3 4 2 5 3 7	0 2 4 6 7	6 3 | 4 2 | 5 3 | 7

Formally, given a sequence s as above, we call **monotonic cutpoints** *any* integer sequence

$$cut \quad = \quad c_0\, c_1\, \ldots\, c_{m-1}$$

such that the following four properties hold:

[1] The topic of this challenge was suggested by Nadia Polikarpova.

non-empty: $m > 0$

begin-to-end: $c_0 = 0$ and $c_{m-1} = n$

within bounds: for every element $c_k \in cut$: $0 \leq c_k \leq n$

monotonic: for every pair of consecutive elements $c_k, c_{k+1} \in cut$, the segment

$$s[c_k..c_{k+1}) \quad = \quad s[c_k]\, s[c_k + 1] \ldots s[c_{k+1} - 1]$$

of s, which starts at index c_k included and ends at index c_{k+1} excluded, is *monotonic*, that is:

1. either $s[c_k] < s[c_k + 1] < \cdots < s[c_{k+1} - 1]$
2. or $s[c_k] \geq s[c_k + 1] \geq \cdots \geq s[c_{k+1} - 1]$

(If you prefer, you can change the definition of *monotonic* so that segments of equal values can be indifferently included in increasing or in decreasing segments. If you choose to do so, you may have to change the algorithm given below to match your definition of monotonic segment.)

In this challenge we focus on **maximal** monotonic cutpoints, that is such that, if we extend any segment by one element, the extended segment is not monotonic anymore.

Given a sequence s, for example stored in an array, maximal monotonic cutpoints can be computed by scanning s once while storing every index that corresponds to a change in monotonicity (from increasing to decreasing, or vice versa), as shown in the following algorithm.

```
cut := [0]  # singleton sequence with element 0
x, y := 0, 1
while y < n:   # n is the length of sequence s
   increasing := s[x] < s[y]  # currently in increasing segment?
   while y < n and (s[y-1] < s[y]) == increasing:
      y := y + 1
   cut.extend(y)  # extend cut by adding y to its end
   x := y
   y := x + 1
if x < n:
   cut.extend(n)
```

Tasks

Implementation task. Implement the algorithm shown above to compute monotonic cutpoints of an input sequence. Choose any representation of input sequence and cutpoints sequence that is manageable using your programming language of choice: arrays, mathematical sequences, dynamic lists, ... If you are using a functional programming language, you can use recursion instead of looping to implement the general idea behind the algorithm.

Verification tasks.

1. Verify that the output sequence satisfies properties *non-empty*, *begin-to-end*, and *within bounds* above.
2. Verify that the output sequence satisfies property *monotonic* given above (*without* the maximality requirement).
3. **Optional task (advanced)**: Strengthen the definition of monotonic cutpoints so that it requires *maximal* monotonic cutpoints, and prove that your algorithm implementation computes maximal cutpoints according to the strengthened definition.

2 Part B: GHC Sort

The GHC Haskell compiler's standard library includes an implementation of a generic sorting method which is a form of *patience sorting*.[2] To sort a sequence s, **GHC sort** works as follows:

1. Split s into monotonic segments $\sigma_1, \sigma_2, \ldots, \sigma_{m-1}$
2. Reverse every segment that is decreasing
3. Merge the segments *pairwise* in a way that preserves the order
4. If all segments have been merged into one, that is an ordered copy of s; then terminate. Otherwise, go to step 3

Merging in step 3 works like merging in Merge Sort:

```
# merge ordered segments s and t
merged := []
x, y := 0, 0
while x < length(s) and y < length(t):
  if s[x] < t[y]:
    merged.extend(s[x])
    x := x + 1
  else:
    merged.extend(t[y])
    y := y + 1
# append any remaining tail of s or t
while x < length(s):
  merged.extend(s[x])
  x := x + 1
while y < length(t):
  merged.extend(t[y])
  y := y + 1
```

For example, GHC sort applied to the sequence $s = 3\ 2\ 8\ 9\ 3\ 4\ 5$ goes through the following steps:

- monotonic segments: 3 2 | 8 9 | 3 4 5

[2] Named after the patience card game https://en.wikipedia.org/wiki/Patience_sorting.

- reverse decreasing segments: 2 3 | 8 9 | 3 4 5
- merge segments pairwise: 2 3 8 9 | 3 4 5
- merge segments pairwise again: 2 3 3 4 5 8 9, which is s sorted

Tasks

Implementation task. Implement GHC sort in your programming language of choice. Again, you can represent the sequences of ordered segments using any data structure or abstract representation that works well with your tool.

To compute the monotonic segments of the input you can rely on the algorithm developed in part 1 of this challenge. If you find it preferable, you can add the reversal (step 2) to the same pass that constructs the monotonic segments in step 1.

Verification tasks.

1. Write functional specifications of all procedures/functions/main steps of your implementation.
2. Verify that the implementation of *merge* returns a sequence `merged` that is **sorted**.
3. Verify that the overall sorting algorithm returns an output that is sorted.
4. Verify that the overall sorting algorithm returns an output that is a permutation of the input.

According to the capabilities of your verification tool, you may focus on the parts of the algorithm that are more amenable to analysis, while specifying the expected behavior of the other parts without proving their correctness explicitly.