Challenge 2
Mergesort with Runs

VerifyThis at ETAPS 2022
Organizers: Marie Farrell, Peter Lammich
Steering Committee: Marieke Huisman, Rosemary Monahan, Peter Müller, Mattias Ulbrich

2–3 April 2022, TU Munich, Germany

Disclaimer: the programs may contain bugs. If you find any, fix them and mention the fix in your submission!

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

Problem Description

The following mergesort based algorithm sorts the elements of a list or array, and, at the same time, computes the indexes where runs of equal elements end.

The program below operates on a type T with a *weak partial ordering* $\leq$, i.e., there exists a mapping $f : T \to L$, such that L is linearly ordered and $t_1 \leq t_2$ iff $f(t_1) \leq f(t_2)$. Moreover, we assume an unsigned integer type `size_t` large enough to hold array indexes, and a type `array` that supports indexing `a[i]` and pushing elements to the back `a.push_back`, extending the array's size. Arrays are always initialized as empty array with length zero.

Example

```
msort([5,4,5,3,9,3],0,6) = {
  runs: [    2,3,  5,6] // end indexes or runs (exclusive)
  data: [3,3,4,5,5,9]   // sorted input
}
```

```
// Structure to store sorted array and end indexes of runs
struct sr {
  array<size_t> runs; // End indexes of runs (exclusive)
  array<T> data; // Data
}

// Merge r1 and r2
sr merge(sr r1, sr r2) {
  sr res;
  size_t di1 = 0; size_t di2 = 0; // Current positions in data arrays
  size_t ri1 = 0; size_t ri2 = 0; // Current positions in runs

  while (ri1 < r1.runs.length or ri2 < r2.runs.length) {
    // Check if we have to take data from first and/or second input array
    bool t1 = ri1 < r1.runs.length
             and (ri2 == r2.runs.length or r1.data[di1] <= r2.data[di2]);
    bool t2 = ri2 < r2.runs.length
             and (ri1 == r1.runs.length or r2.data[di2] <= r1.data[di1]);

    if (t1) { // Copy data from first input array
      for (;di1 < r1.runs[ri1]; ++di1 ) res.data.push_back(r1.data[di1]);
      ++ri1;
    }

    if (t2) { // Copy data from second input array
      for (;di2 < r2.runs[ri2]; ++di2) res.data.push_back(r2.data[di2]);
      ++ri2;
    }

    // Add new segment boundary
    res.runs.push_back(res.data.size());
  }

  return res;
}

// Mergesort array in between l and h. assumes l<=h
sr msort(array<T> a, size_t l, size_t h) {
  // Corner cases
  if (l == h) return res;
  if (h - l == 1) {
    res.data.push_back(a[l]);
    res.runs.push_back(res.data.size());
    return res;
  }

  size_t m = l + (h - l)/2; // Compute middle index

  sr res1 = msort(a,l,m);   // Sort left side
  sr res2 = msort(a,m,h);   // Sort right side
  return merge(res1,res2);   // Merge
}
```

Tasks

Implementation task. Implement the `merge` and `msort` functions. Implement the `array`, `T`, and `size_t` types in any way that fits the tool you are using.

If you cannot handle fixed bit-width types, you may use arbitrary precision integers for `size_t`. If you cannot handle generic types or type-classes, you may fix `T` to some concrete type, e.g., integer.

Verification tasks. Verify the following properties:

1. memory safety
2. termination
3. `merge` merges correctly (permutation and sortedness)
4. `merge` returns the correct run indexes
5. `msort` sorts the input (permutation and sortedness)
6. `msort` returns the correct run indexes
7. `msort` is a stable sorting algorithm
8. `msort` runs in $O(n \log n)$ time and $O(n \log n)$ space.