```rust
#[inline]
fn sort_standard_wip(
   sorting_data: &mut [SortableEntity],
   sorting_stats: &mut SortStats,
   config: WipSortConfig,
) -> usize {
   let stack_size = std::mem::size_of::<usize>() * 8 + 1;
   // ...
   sorting_stats.start_one();
   for merge_step in iter {
      sorting_stats.start_subsort(merge_step.shifted_level);

      if merge_step.singles_to_add > 0 {
         // ...
      }
      assert!(subsort_stack.len() <= stack_size);

      let right_sub = subsort_stack.pop().unwrap();
      let left_sub = subsort_stack.pop().unwrap();
      let right_tail = right_sub.tail;
      let left_tail = left_sub.tail;
      let right_p_dfs_start = right_sub.p_dfs_start;
      // let left_p_dfs_start = left_sub.p_dfs_start;
      // let right_c_dfs_start = right_sub.c_dfs_start;
      let left_c_dfs_start = left_sub.c_dfs_start;

      // Consider these assignments fake. Really Rust should allow
      // guaranteed assignment in conditional that follows.

      let final_head: usize;

      let mut curr_head: usize;

      let right_head = right_sub.head;
      let left_head = left_sub.head;
      let mut curr_b: usize = right_head;
      let mut curr_a: usize = left_head;

      sorting_data[left_tail].set_forward_link(Some(right_head));
      sorting_data[right_tail].set_forward_link(Some(left_head));

      // INCR.
      // Should update to n stat.
      sorting_stats.increment_nlogn(merge_step.shifted_level);
      let mut consume_left: bool = sorting_data[curr_a] <= sorting_data[curr_b];
      if consume_left {
         final_head = curr_a;
         curr_head = right_tail;
      } else {
         final_head = curr_b;
         curr_head = left_tail;
      }

      // Rust cannot cope with speed and cannot allow for reasonable declaration of
 disjointedness.
      //
      // let left_min = &sorting_data[left_head];
      // let right_min = &sorting_data[right_head];
      // let left_max = &sorting_data[left_tail];
      // let right_max = &sorting_data[right_tail];

      // ========================================
```

```
      // Surgery on fronts.

      if config.upper_lozenge {
         let mut trimming_tr = Some(merge_step.middle - 1);
         while (trimming_tr != None)
            && (sorting_data[trimming_tr.unwrap()] <= sorting_data[right_tail])
         {
            trimming_tr = sorting_data[trimming_tr.unwrap()].get_tertiary_link();
         }
         sorting_data[right_tail].set_tertiary_link(trimming_tr);

         let mut trimming_tl = Some(merge_step.middle);
         while (trimming_tl != None)
            && (sorting_data[trimming_tl.unwrap()] < sorting_data[left_tail])
         {
            trimming_tl = sorting_data[trimming_tl.unwrap()].get_secondary_link();
         }
         sorting_data[left_tail].set_secondary_link(trimming_tl);
      }

      //

      let mut lozenge_br;
      let mut trimming_br;
      if sorting_data[merge_step.middle - 1] > sorting_data[right_head] {
         trimming_br = left_c_dfs_start;
         lozenge_br = Some(merge_step.middle - 1);
         while (trimming_br != None)
            && (sorting_data[trimming_br.unwrap()] > sorting_data[right_head])
         {
            let new_trimming_br = sorting_data[trimming_br.unwrap()].get_tertiary_l
ink();
            sorting_data[trimming_br.unwrap()].set_tertiary_link(lozenge_br);

            lozenge_br = trimming_br;
            trimming_br = new_trimming_br;
         }
      } else {
         trimming_br = Some(merge_step.middle - 1);
         lozenge_br = None;
         // When appending, the appended part needs to be complete chain.
         sorting_data[merge_step.middle - 1].set_tertiary_link(left_c_dfs_start);
      }
      // Save R block head, and link to current. Probably just compare with NULL.
      let orig_right_px_dfs_start = sorting_data[merge_step.upper - 1].get_tertiary
_link();
      sorting_data[merge_step.upper - 1].set_tertiary_link(right_sub.c_dfs_start);
      // Append, which may be to head.
      sorting_data[right_head].set_tertiary_link(trimming_br);
      // Extract current head and restore original.
      let right_c_dfs_start = sorting_data[merge_step.upper - 1].get_tertiary_link(
);
      sorting_data[merge_step.upper - 1].set_tertiary_link(orig_right_px_dfs_start)
;

      //

      let mut lozenge_bl;
      let mut trimming_bl;
      if sorting_data[merge_step.middle] >= sorting_data[left_head] {
         trimming_bl = right_p_dfs_start;
         lozenge_bl = Some(merge_step.middle);
```

```
         while (trimming_bl != None)
            && (sorting_data[trimming_bl.unwrap()] >= sorting_data[left_head])
         {
            let new_trimming_bl = sorting_data[trimming_bl.unwrap()].get_secondary_
link();
            sorting_data[trimming_bl.unwrap()].set_secondary_link(lozenge_bl);

            lozenge_bl = trimming_bl;
            trimming_bl = new_trimming_bl;
         }
      } else {
         trimming_bl = Some(merge_step.middle);
         lozenge_bl = None;
         // When appending, the appended part needs to be complete chain.
         sorting_data[merge_step.middle].set_secondary_link(right_p_dfs_start);
      }
      // Save R block head, and link to current. Probably just compare with NULL.
      let orig_left_cx_dfs_start = sorting_data[merge_step.lower].get_secondary_lin
k();
      sorting_data[merge_step.lower].set_secondary_link(left_sub.p_dfs_start);
      // Append, which may be to head.
      sorting_data[left_head].set_secondary_link(trimming_bl);
      // Extract current head and restore original.
      let left_p_dfs_start = sorting_data[merge_step.lower].get_secondary_link();
      sorting_data[merge_step.lower].set_secondary_link(orig_left_cx_dfs_start);

      //

      // trimming_br is just below lowest in right, or None.
      // trimming_bl is just below lowest in left, or None.

      // Done with surgery on fronts.
      // ========================================

      // Invariant made into more specific assurance:
      assert!((curr_a != right_head) && (curr_b != left_head));

      while (curr_a != right_head) && (curr_b != left_head) {
         // INCR.
         sorting_stats.increment_nlogn(merge_step.shifted_level);
         consume_left = sorting_data[curr_a] <= sorting_data[curr_b];
         if consume_left {
            sorting_data[curr_head].set_forward_link(Some(curr_a));
            sorting_data[curr_a].set_backward_link(Some(curr_head));
            curr_head = curr_a;
            curr_a = sorting_data[curr_a].get_forward_link().unwrap();
         } else {
            sorting_data[curr_head].set_forward_link(Some(curr_b));
            sorting_data[curr_b].set_backward_link(Some(curr_head));
            curr_head = curr_b;
            curr_b = sorting_data[curr_b].get_forward_link().unwrap();
         }

         // Invariant:
         assert!(
            (consume_left && (curr_b != left_head)) || (!consume_left && (curr_a !=
 right_head))
         );
      }

      // Invariant made into more specific assurance.
      assert!(
```

```
         (consume_left && (curr_b != left_head) && (curr_a == right_head))
            || (!consume_left && (curr_a != right_head) && (curr_b == left_head))
      );

      // Need to consume opposite of what was last consumed.
      let final_tail: usize = if consume_left {
         sorting_data[curr_head].set_forward_link(Some(curr_b));
         sorting_data[curr_b].set_backward_link(Some(curr_head));
         right_tail
      } else {
         sorting_data[curr_head].set_forward_link(Some(curr_a));
         sorting_data[curr_a].set_backward_link(Some(curr_head));
         left_tail
      };

      // Restore ends of linked list of sorted nodes.
      sorting_data[final_tail].set_forward_link(None);
      sorting_data[final_head].set_backward_link(None);

      // println!("{} + < {}-{} >", subsort_stack.len(), merge_step.lower, merge_st
ep.upper);
      subsort_stack.push(WipSubSort {
         has_reverse: true,
         head: final_head,
         tail: final_tail,
         // p_dfs_start: None,
         // c_dfs_start: None,
         p_dfs_start: left_p_dfs_start,
         c_dfs_start: right_c_dfs_start,
      });
      sorting_stats.finish_subsort(merge_step.shifted_level);
   }

   sorting_stats.finish_one();

   let final_subsort = subsort_stack.pop().unwrap();

   // Final subsort is the completed sort.
   final_subsort.head
}
```