Sorting Programs Executing Fewer Branches

Jyrki Katajainen

Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark

Abstract. When sorting a sequence of n elements, it is well-known that every comparison-based sorting algorithm must perform at least $n \lg n - O(n)$ element comparisons. Hence, it seems unavoidable that any sorting algorithm will also execute about the same number of conditional branches. Since nothing is known about the order of the input elements, it is hard to predict the outcome of the branches related to element comparisons. On an average, every second such branch may lead to a branch misprediction.

In this report, together with an accompanying `tar` ball, we release the source code of the sorting programs discussed and experimented in the following papers:

[1] Amr Elmasry, Jyrki Katajainen, and Max Stenmark, Branch mispredictions don't affect mergesort, *Proceedings of the 11th International Symposium on Experimental Algorithms*, Lecture Notes in Computer Science **7276**, Springer-Verlag (2012), 160-171

[2] Amr Elmasry and Jyrki Katajainen, Lean programs, branch mispredictions, and sorting, *Proceedings of the 6th International Conference on Fun with Algorithms*, Lecture Notes in Computer Science **7288**, Springer-Verlag (2012), 119-130

The key idea in the described sorting programs is to decouple element comparisons from conditional branches. Therefore, these programs will incur significantly fewer branch mispredictions than, for example, the C++ standard-library introsort (introspective median-of-three quicksort that switches to heapsort if the recursion depth gets too high).

The sorting programs given can be divided into three categories depending on how many branch mispredictions they incur during their execution.

Lean: $O(1)$ branch mispredictions incurred when the branch predictor used by the underlying hardware is static.

Moderately optimized: $O(n)$ branch mispredictions incurred under the same assumptions as above.

Unoptimized: The number of branch mispredictions incurred is proportional to the number of element comparisons performed.

As shown in [2], all programs can be made lean. However, in practice, a moderately-optimized variant is often faster than a lean variant. This is also true for the variants of heapsort, mergesort, and quicksort studied here.

Keywords. Sorting, branch prediction, branchless code, lean programs

CPH STL Report 2014-1, December 2014. Revised January 2018.

Release date

2018-01-31

Table of contents

§ 1 Preliminaries

In [1,2] (the list of references is given in the abstract), in the theoretical analyses we assumed that the branch predictor used by the underlying hardware is static. A typical static predictor assumes that forward branches are not taken and backward branches are taken.

Definition 1. *A program is said to be* lean *if it has only a constant number of conditional branches.*

When the loops are branchless, except the final conditional branch at the end, and when there are only a constant number of unnested branchless loops, a static branch predictor can process the program such that at most $O(1)$ branch mispredictions will be incurred.

According to Oxford Advanced Learner's Dictionary, *in situ* means "in the original or correct place". Be aware that, when we refer to the amount of memory used by a program, we make a distinction between the concepts *in place* and *in situ* even though semantically they mean the same.

Definition 2. *A program is said to run* in place *if, in addition to the input, it uses $O(1)$ words of memory and never stores more than $O(1)$ elements outside the input sequence.*

Definition 3. *Assume that the size of the input is n. A program is said to run* in situ *if, in addition to the input, it uses $O(\lg n)$ words of memory and never stores more than $O(1)$ elements outside the input sequence.*

Intentionally, we kept the names of template parameters short in our programs. Their meaning is the following:

- `C`: the type of a comparator used in element comparisons
- `D`: the type of the distance between two iterators
- `N`: the integer type specifying a size
- `V`: the type of the elements in the input sequence
- `R`: the type of a random-access iterator specifying a position of an element in the input sequence, normally the first or one-past-the-end element.

In the documentation comments of the described programs, some numbers are given that indicate the practical performance on a personal computer (Janus) when sorting a random permutation of integers $\{0, 1, \ldots, n-1\}$ for $n \in \{2^{10}, 2^{15}, 2^{20}, 2^{25}\}$. The environment, where these simple experiments were run, was the following:

processor: Intel® Core™ i5-2520M CPU @ 2.50GHz × 4
word size: 64 bits
main memory: 3.8 GB
L3 cache: 3 MB, 12-way associative
cache line: 64 B
operating system: Ubuntu 14.04 LTS
Linux kernel: 3.13.0-40-generic
compiler: `g++` version 4.8.2
compiler options: `-O3 -Wall -std=c++11`

§ 2 **Heapsort**

floyd.h++

```
/*
   This program is a translation of Floyd's original Algol program
   into C++.

   Source: Robert W. Floyd, Algorithm 245: Treesort 3, Communications
   of the ACM 7,12 (1964), 701

   Author: Jyrki Katajainen © 1999, 2011

   Worst–case running time: O(n lg n)
   # element comparisons: 2n lg n + O(n)
   # element assignments: n lg n + O(n)
   # branch mispredictions: 0.5 n lg n + O(n) [expected]
   Extra space: O(1) elements, O(1) words

   Observed sorting time per n lg n [ns]; input: random permutation
       1024      6.25
      32768      6.21
    1048576      7.22
   33554432     12.9
*/

#include <algorithm> // std::iter_swap
#include <iterator> // std::iterator_traits

namespace floyd {

  template<typename R, typename D, typename C>
  void sift_down(R a, D i, D n, C less) {
    using V = typename std::iterator_traits<R>::value_type;
    D j;
    V copy = a[i];
  loop:
    j = 2 * i;
    if (j ≤ n) {
      if (j < n) {
        if (less(a[j], a[j + 1])) {
          j = j + 1;
        }
      }
      if (less(copy, a[j])) {
        a[i] = a[j];
        i = j;
        goto loop;
      }
    }
    a[i] = copy;
```

```
48    }
49
50    template<typename R, typename C>
51    void make_heap(R first, R past, C less) {
52      using D = typename std::iterator_traits<R>::difference_type;
53      R const a = first - 1;
54      D const n = past - first;
55      for (D i = n / 2; i > 0; --i) {
56        sift_down(a, i, n, less);
57      }
58    }
59
60    template<typename R, typename C>
61    void sort_heap(R first, R past, C less) {
62      using D = typename std::iterator_traits<R>::difference_type;
63      R const a = first - 1;
64      D const n = past - first;
65      for (D i = n; i >= 2; --i) {
66        std::iter_swap(a + 1, a + i);
67        sift_down(a, D(1), i - 1, less);
68      }
69    }
70
71    template<typename R, typename C>
72    void sort(R a, R b, C less) {
73      floyd::make_heap(a, b, less);
74      floyd::sort_heap(a, b, less);
75    }
76  }
```

williams.h++

```
1   /*
2      This program is a translation of Williams' original Algol program
3      into C++.
4
5      Source: J.W.J. Williams, Algorithm 232, Heapsort, Communications
6      of the ACM 7,6 (1964) 347–348
7
8      Author: Jyrki Katajainen © 1999, 2011
9
10     Worst–case running time: O(n lg n)
11    # element comparisons: 3n lg n + O(n)
12    # element assignments: n lg n + O(n)
13    # branch mispredictions: 0.5 n lg n + O(n) [expected]
14     Extra space: O(1) elements, O(1) words
15
16     Observed sorting time per n lg n [ns]; input: random permutation
17         1024      6.32
18        32768      6.28
19      1048576      7.42
```

```
20     33554432     13.3
21 */
22
23 #include <iterator> // std::iterator_traits
24
25 namespace williams {
26
27   template<typename R, typename D, typename C>
28   void sift_up(R a, D j, C less) {
29     using V = typename std::iterator_traits<R>::value_type;
30     V in = a[j];
31   scan:
32     if (j > 1) {
33       D i = j / 2;
34       if (less(a[i], in)) {
35         a[j] = a[i];
36         j = i;
37         goto scan;
38       }
39     }
40     a[j] = in;
41   }
42
43   template<typename R, typename C>
44   void make_heap(R first, R past, C less) {
45     using D = typename std::iterator_traits<R>::difference_type;
46     D const n = past - first;
47     if (n < 2) {
48       return;
49     }
50     R const a = first - 1;
51     D j = 1;
52   inheap:
53     j = j + 1;
54     sift_up(a, j, less);
55     if (j < n) goto inheap;
56   }
57
58   template<typename R, typename D, typename V, typename C>
59   void sift_down(R a, D n, V in, C less) {
60     D i = 1;
61   loop:
62     D j = i + i;
63     if (j ≤ n) {
64       if (less(a[j], a[j + 1])) {
65         j += 1;
66       }
67       if (less(in, a[j])) {
68         a[i] = a[j];
69         i = j;
70         goto loop;
71       }
```

```
72      }
73      a[i] = in;
74    }
75
76    template<typename R, typename C>
77    void sort_heap(R first, R past, C less) {
78      using D = typename std::iterator_traits<R>::difference_type;
79      using V = typename std::iterator_traits<R>::value_type;
80      D const n = past - first;
81      if (n < 2) {
82        return;
83      }
84      R const a = first - 1;
85      D i = n;
86    outheap:
87      V out = a[1];
88      V in = a[i];
89      sift_down(a, i - 1, in, less);
90      a[i] = out;
91      i = i - 1;
92      if (i > 1) goto outheap;
93    }
94
95    template<typename R, typename C>
96    void sort(R a, R b, C less) {
97      williams::make_heap(a, b, less);
98      williams::sort_heap(a, b, less);
99    }
100 }
```

optimized_williams.h++

```
1  /*
2     This program is as Williams' original except the following change
3     in sift_down():
4
5     <       if (less(a[j], a[j + 1])) {
6     <         j += 1;
7     <       }
8     ---
9     >       j += less(a[j], a[j + 1]);
10
11    Source: Amr Elmasry and Jyrki Katajainen, Lean programs, branch
12    mispredictions, and sorting, Proceedings of the 6th International
13    Conference on Fun with Algorithms, Lecture Notes in Computer
14    Science 7288, Springer–Verlag (2012), 119–130
15
16    Author: Jyrki Katajainen © 2011
17
18    Worst–case running time: O(n lg n)
19    # element comparisons: 3n lg n + O(n)
```

```
20     # element assignments: n lg n + O(n)
21     # branch mispredictions: O(n)
22     Extra space: O(1) elements, O(1) words
23
24     Observed sorting time per n lg n [ns]; input: random permutation
25         1024      3.72
26        32768      3.97
27      1048576      7.37
28     33554432     26.5
29  */
30
31  #include <iterator> // std::iterator_traits
32
33  namespace optimized_williams {
34
35    template<typename R, typename D, typename C>
36    void sift_up(R a, D j, C less) {
37      using V = typename std::iterator_traits<R>::value_type;
38      V in = a[j];
39    scan:
40      if (j > 1) {
41        D i = j / 2;
42        if (less(a[i], in)) {
43          a[j] = a[i];
44          j = i;
45          goto scan;
46        }
47      }
48      a[j] = in;
49    }
50
51    template<typename R, typename C>
52    void make_heap(R first, R past, C less) {
53      using D = typename std::iterator_traits<R>::difference_type;
54      D const n = past - first;
55      if (n < 2) {
56        return;
57      }
58      R const a = first - 1;
59      D j = 1;
60    inheap:
61      j = j + 1;
62      sift_up(a, j, less);
63      if (j < n) goto inheap;
64    }
65
66    template<typename R, typename D, typename V, typename C>
67    void sift_down(R a, D n, V in, C less) {
68      D i = 1;
69    loop:
70      D j = i + i;
71      if (j ≤ n) {
```

```
72        j += less(a[j], a[j + 1]);
73        if (less(in, a[j])) {
74          a[i] = a[j];
75          i = j;
76          goto loop;
77        }
78      }
79      a[i] = in;
80    }
81
82    template<typename R, typename C>
83    void sort_heap(R first, R past, C less) {
84      using D = typename std::iterator_traits<R>::difference_type;
85      using V = typename std::iterator_traits<R>::value_type;
86      D const n = past - first;
87      if (n < 2) {
88        return;
89      }
90      R const a = first - 1;
91      D i = n;
92    outheap:
93      V out = a[1];
94      V in = a[i];
95      sift_down(a, i - 1, in, less);
96      a[i] = out;
97      i = i - 1;
98      if (i > 1) goto outheap;
99    }
100
101   template<typename R, typename C>
102   void sort(R a, R b, C less) {
103     optimized_williams::make_heap(a, b, less);
104     optimized_williams::sort_heap(a, b, less);
105   }
106 }
```

lean_heapsort.h++

```
1  /*
2     In ideal conditions, this heapsort program incurs at most O(1)
3     branch mispredictions. However, we could not force the compiler to
4     translate conditional moves with cmov instructions.
5
6     Source: Amr Elmasry and Jyrki Katajainen, Lean programs, branch
7     mispredictions, and sorting, Proceedings of the 6th International
8     Conference on Fun with Algorithms, Lecture Notes in Computer
9     Science 7288, Springer–Verlag (2012), 119–130
10
11    Author: Jyrki Katajainen © 2011
12
13    Worst–case running time: O(n lg n)
```

```
14    # element comparisons: 2n lg n + O(n)
15    # element assignments: 5n lg n + O(n)
16    # branch mispredictions: O(1)
17    Extra space: O(1) elements, O(1) words
18
19    Observed sorting time per n lg n [ns]; input: random permutation
20        1024     3.86
21       32768     4.56
22     1048576     8.56
23    33554432    27.7
24 */
25
26 #include <algorithm> // std::swap
27 #include <iterator> // std::iterator_traits
28
29 namespace lean_heapsort {
30
31   template<typename R, typename D, typename C>
32   void sift_up(R a, D j, C less) {
33     using V = typename std::iterator_traits<R>::value_type;
34     V in = a[j];
35     while (j > 1) {
36       D i = j / 2;
37       if (less(a[i], in)) {
38         a[j] = a[i];
39         j = i;
40       }
41       else {
42         break;
43       }
44     }
45     a[j] = in;
46   }
47
48   template<typename R, typename C>
49   void make_heap(R first, R past, C less) {
50     using D = typename std::iterator_traits<R>::difference_type;
51     using V = typename std::iterator_traits<R>::value_type;
52     R const a = first - 1;
53     D const n = past - first;
54     D const m = (n bitand 1) ? n : n - 1;
55     D i = m / 2;
56     D j = i;
57     D hole = j;
58     V copy = *first;
59     while (i > 0) {
60       if (i == j) hole = j;
61       if (i == j) copy = a[j];
62       j = 2 * j;
63       j += less(a[j], a[j + 1]);
64       a[hole] = a[j];
65       if (less(copy, a[j])) hole = j;
```

```
66        if (2 * j > m) a[hole] = copy;
67        if (2 * j > m) i = i - 1;
68        if (2 * j > m) j = i;
69      }
70      sift_up(a, n, less);
71    }
72
73    template<typename R, typename C>
74    void sort_heap(R first, R past, C less) {
75      using D = typename std::iterator_traits<R>::difference_type;
76      using V = typename std::iterator_traits<R>::value_type;
77      D n = past - first;
78      if (n < 2) {
79        return;
80      }
81      R const a = first - 1;
82      V out = a[1];
83      V in = a[n];
84      D j = 1;
85      D hole = 1;
86      while (n > 2) {
87        j = 2 * j;
88        j += less(a[j], a[j + 1]);
89        a[hole] = a[j];
90        if (less(in, a[j])) hole = j;
91        bool outer = (2 * j >= n);
92        if (outer) a[hole] = in;
93        if (outer) a[n] = out;
94        if (outer) n = n - 1;
95        if (outer) j = 1;
96        if (outer) out = a[1];
97        if (outer) in = a[n];
98        if (outer) hole = 1;
99      }
100     if (less(a[2], a[1])) {
101       std::swap(a[1], a[2]);
102     }
103   }
104
105   template<typename R, typename C>
106   void sort(R a, R b, C less) {
107     lean_heapsort::make_heap(a, b, less);
108     lean_heapsort::sort_heap(a, b, less);
109   }
110 }
```

§ 3 Mergesort

stable_sort.h++

```
/*
   In the GNU implementation of the C++ standard library (gcc
   version 4.8.2), the algorithm underlying std::stable_sort()
   was bottom-up mergesort.

   Author: Jyrki Katajainen © 2011

   Worst-case running time: O(n lg n)
   # element comparisons: n lg n + O(n)
   # element assignments: n lg n + O(n)
   # branch mispredictions: O(n)
   Extra space: n + O(1) elements, O(1) words

   Observed sorting time per n lg n [ns]; input: random permutation
       1024      3.54
      32768      3.43
    1048576      3.54
   33554432      3.49
*/

#include <algorithm> // std::stable_sort

namespace stable_sort {

  template<typename R, typename C>
  void sort(R a, R b, C less) {
    std::stable_sort(a, b, less);
  }
}
```

in_place_stable_sort.h++

```
/*
   In the GNU implementation of the C++ standard library (gcc
   version 4.8.2), the generic algorithm std::__inplace_stable_sort()
   was an in-place variant of mergesort.

   Author: Jyrki Katajainen © 2011

   Worst-case running time: O(n (lg n)^2)
   Extra space: O(1) elements, O(1) words

   Observed sorting time per n lg n [ns]; input: random permutation
       1024      17.0
      32768      20.5
    1048576      22.9
   33554432      24.8
*/

#include <algorithm> // std::lower_bound std::iter_swap std::rotate
#include "introsort.h++" // introsort::insertion_sort
```

```
20 #include <iterator> // std::advance
21
22 namespace in_place_stable_sort {
23
24   template<typename B, typename C>
25   void merge_without_buffer(B first, B middle, B past, C less) {
26     using D = typename std::iterator_traits<B>::difference_type;
27     D k = std::distance(first, middle);
28     D ℓ = std::distance(middle, past);
29     if (k == 0 or ℓ == 0) {
30       return;
31     }
32     if (k + ℓ == 2) {
33       if (less(*middle, *first)) {
34         std::iter_swap(first, middle);
35       }
36       return;
37     }
38     B first_cut = first;
39     B second_cut = middle;
40     if (k > ℓ) {
41       D k₂ = k / 2;
42       std::advance(first_cut, k₂);
43       second_cut = std::lower_bound(middle, past, *first_cut, less);
44     }
45     else {
46       D ℓ₂ = ℓ / 2;
47       std::advance(second_cut, ℓ₂);
48       first_cut = std::upper_bound(first, middle, *second_cut,
     ↪ less);
49     }
50     std::rotate(first_cut, middle, second_cut);
51     B new_middle = first_cut;
52     std::advance(new_middle, std::distance(middle, second_cut));
53     merge_without_buffer(first, first_cut, new_middle, less);
54     merge_without_buffer(new_middle, second_cut, past, less);
55   }
56
57   template<typename R, typename C>
58   void inplace_stable_sort(R first, R past, C less) {
59     if (past - first < 15) {
60       introsort::insertion_sort(first, past, less);
61       return;
62     }
63     R middle = first + (past - first) / 2;
64     inplace_stable_sort(first, middle, less);
65     inplace_stable_sort(middle, past, less);
66     merge_without_buffer(first, middle, past, less);
67   }
68
69   template<typename R, typename C>
70   void sort(R first, R past, C less) {
```

```
71      inplace_stable_sort(first, past, less);
72    }
73  }
```

tuned_mergesort.h++

```
1   /*
2      This program implements bottom–up mergesort with the following
3      optimization:
4
5      – Instead of using insertionsort, it sorts each chunk of size four
6        with straight–line code that has no conditional branches.
7
8      Source: Amr Elmasry, Jyrki Katajainen, and Max Stenmark, Branch
9      mispredictions don't affect mergesort, Proceedings of the 11th
10     International Symposium on Experimental Algorithms, Lecture Notes
11     in Computer Science 7276, Springer–Verlag (2012), 160–171
12
13     Author: Jyrki Katajainen © 2011
14
15     Worst–case running time: O(n lg n)
16     # element comparisons: n lg n + O(n)
17     # element assignments: n lg n + O(n)
18     # branch mispredictions: O(n)
19     Extra space: n + O(1) elements, O(1) words
20
21     Observed sorting time per n lg n [ns]; input: random permutation
22          1024      3.02
23         32768      3.03
24       1048576      3.15
25      33554432      3.25
26  */
27
28  #include <algorithm> // std::sort std::copy
29  #include <iterator> // std::iterator_traits
30
31  namespace tuned_mergesort {
32
33    template<typename input, typename D, typename C>
34    void two_passes(input a, D n, C less) {
35      using V = typename std::iterator_traits<input>::value_type;
36      D const rest = n bitand 3;
37      input const boundary = a + n - rest;
38      for (input s = a; s < boundary; s += 4) {
39        input u = s;
40        input v = s + 1;
41        input x = s + 2;
42        input y = s + 3;
43        bool c = less(*v, *u);
44        input t = u;
45        u = c ? v : u;
```

```
46        v = c ? t : v;
47        c = less(*y, *x);
48        t = x;
49        x = c ? y : x;
50        y = c ? t : y;
51        c = less(*x, *u);
52        t = u;
53        u = c ? x : u;
54        x = c ? t : x;
55        c = less(*y, *v);
56        t = v;
57        v = c ? y : v;
58        y = c ? t : y;
59        c = less(*v, *x);
60        t = x;
61        x = c ? v : x;
62        v = c ? t : v;
63        V e1 = *u;
64        V e2 = *x;
65        V e3 = *v;
66        V e4 = *y;
67        *s = e1;
68        *(s + 1) = e2;
69        *(s + 2) = e3;
70        *(s + 3) = e4;
71      }
72      std::sort(boundary, boundary + rest, less);
73    }
74
75    template<typename input, typename output, typename D, typename C>
76    input remaining_passes(input a, output b, D n, C less) {
77      D size = 4;
78      D mask = size - 1;
79      while (size < n) {
80        input p = a;
81        output s = b;
82        input r = a + n;
83        while (p + 2 * size < r) {
84          input t1 = p + size;
85          input t2 = p + 2 * size;
86          input q = t1;
87          input t = nullptr;
88          do {
89            if (less(*q, *p)) {
90              t = q++;
91            }
92            else {
93              t = p++;
94            }
95            *s++ = *t++;
96          } while (((t2 - t) bitand mask));
97          q = (t == p) ? q : p;
```

```
98          t = (t == p) ? t2 : t1;
99          do {
100           *s++ = *q++;
101         } while (q != t);
102         p = t2;
103       }
104       while (p + size < r) {
105         input t1 = p + size;
106         input t2 = (t1 + size < r) ? t1 + size : r;
107         input q = t1;
108         while (p < t1 and q < t2) {
109           if (less(*q, *p)) {
110             *s++ = *q++;
111           }
112           else {
113             *s++ = *p++;
114           }
115         }
116         while (p < t1) {
117           *s++ = *p++;
118         }
119         while (q < t2) {
120           *s++ = *q++;
121         }
122         p = t2;
123       }
124       while (p < r) {
125         *s++ = *p++;
126       }
127       size = size << 1;
128       mask = size - 1;
129       input c = a;
130       a = b;
131       b = c;
132     }
133     return a;
134   }
135
136   template<typename R, typename C>
137   void sort(R first, R past, C less) {
138     using V = typename std::iterator_traits<R>::value_type;
139     using D = typename std::iterator_traits<R>::difference_type;
140     D const n = past - first;
141     V* tmp = new V[n];
142     two_passes(first, n, less);
143     V* x = remaining_passes(first, tmp, n, less);
144     if (x == tmp) {
145       std::copy(tmp, tmp + n, first);
146     }
147     delete[] tmp;
148   }
149 }
```

unrolled_mergesort.h++

```
/*
   This program implements bottom–up mergesort with the following two
   optimizations:

   – Instead of using insertionsort, it sorts each chunk of size four
     with straight–line code that has no conditional branches.

   – In the main loop of the merge routine, it moves four elements to
     the output area in each iteration.

   Source: Amr Elmasry, Jyrki Katajainen, and Max Stenmark, Branch
   mispredictions don't affect mergesort, Proceedings of the 11th
   International Symposium on Experimental Algorithms, Lecture Notes
   in Computer Science 7276, Springer–Verlag (2012), 160–171

   Author: Jyrki Katajainen © 2011

   Worst–case running time: O(n lg n)
   # element comparisons: n lg n + O(n)
   # element assignments: n lg n + O(n)
   # branch mispredictions: O(n)
   Extra space: n + O(1) elements, O(1) words

   Observed sorting time per n lg n [ns]; input: random permutation
       1024      2.99
      32768      3.05
    1048576      3.18
   33554432      3.27
*/

#include <algorithm> // std::sort std::copy
#include <iterator> // std::iterator_traits

namespace unrolled_mergesort {

  template<typename input, typename D, typename C>
  void two_passes(input a, D n, C less) {
    using V = typename std::iterator_traits<input>::value_type;
    D const rest = n bitand 3;
    input const boundary = a + n - rest;
    for (input s = a; s < boundary; s += 4) {
      input u = s;
      input v = s + 1;
      input x = s + 2;
      input y = s + 3;
      bool c = less(*v, *u);
      input t = u;
      u = c ? v : u;
      v = c ? t : v;
```

```
50        c = less(*y, *x);
51        t = x;
52        x = c ? y : x;
53        y = c ? t : y;
54        c = less(*x, *u);
55        t = u;
56        u = c ? x : u;
57        x = c ? t : x;
58        c = less(*y, *v);
59        t = v;
60        v = c ? y : v;
61        y = c ? t : y;
62        c = less(*v, *x);
63        t = x;
64        x = c ? v : x;
65        v = c ? t : v;
66        V e1 = *u;
67        V e2 = *x;
68        V e3 = *v;
69        V e4 = *y;
70        *s = e1;
71        *(s + 1) = e2;
72        *(s + 2) = e3;
73        *(s + 3) = e4;
74      }
75      std::sort(boundary, boundary + rest, less);
76    }
77
78    template<typename input, typename output, typename D, typename C>
79    input remaining_passes(input a, output b, D n, C less) {
80      D size = 4;
81      while (size < n) {
82        input p = a;
83        output s = b;
84        input r = a + n;
85        while (p + size < r) {
86          input t1 = p + size;
87          input t2 = (t1 + size < r) ? t1 + size : r;
88          input q = t1;
89          t1 -= 4;
90          t2 -= 4;
91          while (p < t1 and q < t2) {
92            if (less(*q, *p)) {
93              *s++ = *q++;
94            }
95            else {
96              *s++ = *p++;
97            }
98            if (less(*q, *p)) {
99              *s++ = *q++;
100           }
101           else {
```

```
102             *s++ = *p++;
103           }
104           if (less(*q, *p)) {
105             *s++ = *q++;
106           }
107           else {
108             *s++ = *p++;
109           }
110           if (less(*q, *p)) {
111             *s++ = *q++;
112           }
113           else {
114             *s++ = *p++;
115           }
116         }
117         t1 += 4;
118         t2 += 4;
119         while (p < t1 and q < t2) {
120           if (less(*q, *p)) {
121             *s++ = *q++;
122           }
123           else {
124             *s++ = *p++;
125           }
126         }
127         while (p < t1) {
128           *s++ = *p++;
129         }
130         while (q < t2) {
131           *s++ = *q++;
132         }
133         p = t2;
134       }
135       while (p < r) {
136         *s++ = *p++;
137       }
138       size = size << 1;
139       input c = a;
140       a = b;
141       b = c;
142     }
143     return a;
144   }
145
146   template<typename R, typename C>
147   void sort(R first, R past, C less) {
148     using V = typename std::iterator_traits<R>::value_type;
149     using D = typename std::iterator_traits<R>::difference_type;
150     D const n = past - first;
151     V* tmp = new V[n];
152     two_passes(first, n, less);
153     V* x = remaining_passes(first, tmp, n, less);
```

```
154       if (x == tmp) {
155         std::copy(tmp, tmp + n, first);
156       }
157       delete[] tmp;
158     }
159   }
```

in_situ_mergesort.h++

```
1   /*
2     This program sorts half of its input using the other half as its
3     working area and repeats this until there are only O(n/lg n)
4     elements left, which can be sorted using any space–efficient
5     sorting method.
6
7     Source: Amr Elmasry, Jyrki Katajainen, and Max Stenmark, Branch
8     mispredictions don't affect mergesort, Proceedings of the 11th
9     International Symposium on Experimental Algorithms, Lecture Notes
10    in Computer Science 7276, Springer–Verlag (2012), 160–171
11
12    Author: Jyrki Katajainen © 2011
13
14    Worst–case running time: O(n lg n)
15    # element comparisons: n lg n + O(n)
16    # element assignments: 3n lg n + O(n)
17    # branch mispredictions: O(n)
18    Extra space: O(1) elements, O(lg n) words
19
20    Observed sorting time per n lg n [ns]; input: random permutation
21        1024     4.36
22       32768     4.34
23     1048576     4.42
24    33554432     4.54
25  */
26
27  #include <algorithm> // std::sort std::nth_element std::swap
        ↪ std::iter_swap
28  #include <cmath> // ilogb
29
30  namespace in_situ_mergesort {
31
32    template<typename C>
33    class converse_relation {
34    protected:
35      C less;
36
37    public:
38      using first_argument_type = typename C::second_argument_type;
39      using second_argument_type = typename C::first_argument_type;
40      using result_type = bool;
41
```

```
42      explicit converse_relation(C const& less)
43        : less(less) {
44      }
45
46      bool operator()(typename C::first_argument_type const& x,
47                      typename C::second_argument_type const& y)
        ↪ const {
48        return less(y, x);
49      }
50    };
51
52    template<typename input, typename D, typename C>
53    void two_passes(input a, D n, C less) {
54      using V = typename std::iterator_traits<input>::value_type;
55      D rest = n bitand 3;
56      input const boundary = a + n - rest;
57      for (input s = a; s < boundary; s += 4) {
58        input u = s;
59        input v = s + 1;
60        input x = s + 2;
61        input y = s + 3;
62        bool c = less(*v, *u);
63        input t = u;
64        u = c ? v : u;
65        v = c ? t : v;
66        c = less(*y, *x);
67        t = x;
68        x = c ? y : x;
69        y = c ? t : y;
70        c = less(*x, *u);
71        t = u;
72        u = c ? x : u;
73        x = c ? t : x;
74        c = less(*y, *v);
75        t = v;
76        v = c ? y : v;
77        y = c ? t : y;
78        c = less(*v, *x);
79        t = x;
80        x = c ? v : x;
81        v = c ? t : v;
82        V e1 = *u;
83        V e2 = *x;
84        V e3 = *v;
85        V e4 = *y;
86        *s = e1;
87        *(s + 1) = e2;
88        *(s + 2) = e3;
89        *(s + 3) = e4;
90      }
91      std::sort(boundary, boundary + rest, less);
92    }
```

```
93
94  template<typename input, typename output, typename D, typename C>
95  input remaining_passes(input a, output b, D n, C less) {
96    D size = 4;
97    while (size < n) {
98      input p = a;
99      output o = b;
100     input r = a + n;
101     input t1 = nullptr;
102     while (p + size < r) {
103       t1 = p + size;
104       input t2 = (t1 + size < r) ? t1 + size : r;
105       input q = t1;
106       input t = nullptr;
107       while (p < t1 and q < t2) {
108         if (less(*q, *p)) {
109           t = q;
110           ++q;
111         }
112         else {
113           t = p;
114           ++p;
115         }
116         std::iter_swap(t, o++);
117       }
118       while (p < t1) {
119         std::iter_swap(p++, o++);
120       }
121       while (q < t2) {
122         std::iter_swap(q++, o++);
123       }
124       p = t2;
125     }
126     while (p < r) {
127       std::iter_swap(p++, o++);
128     }
129     size = size << 1;
130     std::swap(a, b);
131   }
132   return a;
133 }
134
135 template<typename R, typename C>
136 void mergesort(R p, R r, R t, C less) {
137   using D = typename std::iterator_traits<R>::difference_type;
138   D const n = r - p;
139   two_passes(p, n, less);
140   R q = remaining_passes(p, t, n, less);
141   if (q ≠ t) {
142     while (p ≠ r) {
143       std::iter_swap(p++, t++);
144     }
```

```
145       }
146     }
147
148     template<typename R, typename C>
149     void sort(R p, R r, C less) {
150       using D = typename std::iterator_traits<R>::difference_type;
151       D n = r - p;
152       D const threshold = n / ilogb(2 + n);
153       while (n > threshold) {
154         R q1 = p + n / 2;
155         R q2 = r - n / 2;
156         converse_relation<C> greater(less);
157         std::nth_element(p, q1, r, greater);
158         mergesort(p, q1, q2, less);
159         r = q1;
160         n = r - p;
161       }
162       std::sort(p, r, less);
163     }
164   }
```

median_for_free_in_situ_mergesort.h++

```
1   /*
2       This program sorts half of its input using the other half as its
3       working area and repeats this until there are only O(n/lg n)
4       elements left, which can be sorted using any space–efficient
5       sorting method.
6
7       Warning: This is not a general–purpose sorting routine; it is
8       tuned for sorting only permutations of integers {0,1,...,n–1}.
9
10      Source: Jyrki Katajainen, Branch mispredictions don't affect
11      mergesort, Slides used at the 11th International Symposium on
12      Experimental Algorithms, Bordeaux, 2012–06–08
13
14      Author: Jyrki Katajainen © 2012
15
16      Worst–case running time: ∞
17      Extra space: O(1) elements, O(lg n) words
18
19      Observed sorting time per n lg n [ns]; input: random permutation
20          1024      3.31
21         32768      3.80
22       1048576      4.04
23      33554432      4.26
24  */
25
26  #include <algorithm> // std::sort std::nth_element std::swap
27  #include <cmath> // ilogb
28
```

```
namespace median_for_free_in_situ_mergesort {

  template<typename C>
  class converse_relation {
  protected:
    C less;

  public:
    using first_argument_type = typename C::second_argument_type;
    using second_argument_type = typename C::first_argument_type;
    using result_type = bool;

    explicit converse_relation(C const& less)
      : less(less) {
    }

    bool operator()(typename C::first_argument_type const& x,
        typename C::second_argument_type const& y) const {
      return less(y, x);
    }
  };

  template<typename input, typename D, typename C>
  void two_passes(input a, D n, C less) {
    using V = typename std::iterator_traits<input>::value_type;
    D const rest = n bitand 3;
    input const boundary = a + n - rest;
    for (input s = a; s < boundary; s += 4) {
      input u = s;
      input v = s + 1;
      input x = s + 2;
      input y = s + 3;
      bool c = less(*v, *u);
      input t = u;
      u = c ? v : u;
      v = c ? t : v;
      c = less(*y, *x);
      t = x;
      x = c ? y : x;
      y = c ? t : y;
      c = less(*x, *u);
      t = u;
      u = c ? x : u;
      x = c ? t : x;
      c = less(*y, *v);
      t = v;
      v = c ? y : v;
      y = c ? t : y;
      c = less(*v, *x);
      t = x;
      x = c ? v : x;
      v = c ? t : v;
```

```
81        V e1 = *u;
82        V e2 = *x;
83        V e3 = *v;
84        V e4 = *y;
85        *s = e1;
86        *(s + 1) = e2;
87        *(s + 2) = e3;
88        *(s + 3) = e4;
89      }
90      std::sort(boundary, boundary + rest, less);
91    }
92
93    template<typename input, typename output, typename D, typename C>
94    input remaining_passes(input a, output b, D n, C less) {
95      D size = 4;
96      while (size < n) {
97        input p = a;
98        output o = b;
99        input r = a + n;
100       input t1 = nullptr;
101       while (p + size < r) {
102         t1 = p + size;
103         input t2 = (t1 + size < r) ? t1 + size : r;
104         input q = t1;
105         input t = nullptr;
106         while (p < t1 and q < t2) {
107           if (less(*q, *p)) {
108             t = q;
109             ++q;
110           }
111           else {
112             t = p;
113             ++p;
114           }
115           std::iter_swap(t, o++);
116         }
117         while (p < t1) {
118           std::iter_swap(p++, o++);
119         }
120         while (q < t2) {
121           std::iter_swap(q++, o++);
122         }
123         p = t2;
124       }
125       while (p < r) {
126         std::iter_swap(p++, o++);
127       }
128       size = size << 1;
129       std::swap(a, b);
130     }
131     return a;
132   }
```

```
133
134   template<typename R, typename C>
135   void mergesort(R p, R r, R t, C less) {
136     using D = typename std::iterator_traits<R>::difference_type;
137     D const n = r - p;
138     two_passes(p, n, less);
139     R q = remaining_passes(p, t, n, less);
140     if (q ≠ t) {
141       while (p ≠ r) {
142         std::iter_swap(p++, t++);
143       }
144     }
145   }
146
147   template<typename R, typename V, typename C>
148   void partition(R p, R r, V const& v, C less) {
149     using D = typename std::iterator_traits<R>::difference_type;
150     R q = p;
151     while (q < r and ! less(*q, v)) {
152       ++q;
153     }
154     if (q == r) {
155       return;
156     }
157     std::iter_swap(p, q);
158     ++q;
159     while (q < r) {
160       V x = *q;
161       bool smaller = less(x, v);
162       p += smaller;
163       D delta = smaller * (q - p);
164       R s = p + delta;
165       R t = q - delta;
166       *s = *p;
167       *t = x;
168       ++q;
169     }
170   }
171
172   template<typename R, typename C>
173   void sort(R p, R r, C less) {
174     using D = typename std::iterator_traits<R>::difference_type;
175     using V = typename std::iterator_traits<R>::value_type;
176     D n = r - p;
177     D const threshold = n / (1 * ilogb(2 + n));
178     while (n > threshold) {
179       V median = V((n - 1) / 2);
180       R q1 = p + n / 2;
181       R q2 = r - n / 2;
182       converse_relation<C> greater(less);
183       partition(p, r, median, greater);
184       mergesort(p, q1, q2, less);
```

```
185       r = q1;
186       n = r - p;
187     }
188     std::sort(p, r, less);
189   }
190 }
```

lean_mergesort.h++

```
1  /*
2     In ideal conditions, this mergesort program incurs at most O(1)
3     branch mispredictions. However, we could not force the compiler to
4     translate conditional moves into cmov instructions.
5
6     Source: Amr Elmasry and Jyrki Katajainen, Lean programs, branch
7     mispredictions, and sorting, Proceedings of the 6th International
8     Conference on Fun with Algorithms, Lecture Notes in Computer
9     Science 7288, Springer–Verlag (2012), 119–130
10
11    Author: Jyrki Katajainen © 2011
12
13    Worst–case running time: O(n lg n)
14    # element comparisons: n lg n + O(n)
15    # element assignments: n lg n + O(n)
16    # branch mispredictions: O(1)
17    Extra space: n + O(1) elements, O(1) words
18
19    Observed sorting time per n lg n [ns]; input: random permutation
20        1024     5.92
21       32768     6.21
22     1048576     6.46
23    33554432     6.64
24 */
25
26 #include <algorithm> // std::copy std::sort std::min
27 #include <iterator> // std::iterator_traits
28
29 namespace lean_mergesort {
30
31   template<typename input, typename D, typename C>
32   void two_passes(input a, D n, C less) {
33     using V = typename std::iterator_traits<input>::value_type;
34     D const rest = n bitand 3;
35     input const boundary = a + n - rest;
36     for (input s = a; s < boundary; s += 4) {
37       input u = s;
38       input v = s + 1;
39       input x = s + 2;
40       input y = s + 3;
41       bool c = less(*v, *u);
42       input t = u;
```

```
43        u = c ? v : u;
44        v = c ? t : v;
45        c = less(*y, *x);
46        t = x;
47        x = c ? y : x;
48        y = c ? t : y;
49        c = less(*x, *u);
50        t = u;
51        u = c ? x : u;
52        x = c ? t : x;
53        c = less(*y, *v);
54        t = v;
55        v = c ? y : v;
56        y = c ? t : y;
57        c = less(*v, *x);
58        t = x;
59        x = c ? v : x;
60        v = c ? t : v;
61        V e1 = *u;
62        V e2 = *x;
63        V e3 = *v;
64        V e4 = *y;
65        *s = e1;
66        *(s + 1) = e2;
67        *(s + 2) = e3;
68        *(s + 3) = e4;
69      }
70      std::sort(boundary, boundary + rest, less);
71    }
72
73    template<typename input, typename output, typename D, typename C>
74    input remaining_passes(input a, output b, D n, C less) {
75      D size = 4;
76      D i = 0;
77      D j = 0;
78      D k = 0;
79      D t1 = 0;
80      D t2 = 0;
81      input p = nullptr;
82      D h = 0;
83      while (size < n) {
84        bool next = (k == t2);
85        i = (next) ? t2 : i;
86        t1 = (next) ? std::min(t2 + size, n) : t1;
87        t2 = (next) ? std::min(t1 + size, n) : t2;
88        j = (next) ? t1 : j;
89        h = less(a[j], a[i]) ? j : i;
90        h = (i == t1) ? j : h;
91        h = (j == t2) ? i : h;
92        b[k++] = a[h];
93        i = (h == i) ? i + 1 : i;
94        j = (h == j) ? j + 1 : j;
```

```
95        bool outer = (k == n);
96        size = (outer) ? size << 1 : size;
97        k = (outer) ? 0 : k;
98        t2 = (outer) ? 0 : t2;
99        p = (outer) ? a : p;
100       a = (outer) ? b : a;
101       b = (outer) ? p : b;
102     }
103     return a;
104   }
105
106   template<typename R, typename C>
107   void sort(R first, R past, C less) {
108     using V = typename std::iterator_traits<R>::value_type;
109     using D = typename std::iterator_traits<R>::difference_type;
110     D const n = past - first;
111     V* tmp = new V[n];
112     two_passes(first, n, less);
113     V* x = remaining_passes(first, tmp, n, less);
114     if (x == tmp) {
115       std::copy(tmp, tmp + n, first);
116     }
117     delete[] tmp;
118   }
119 }
```

§ 4 **Quicksort**

std.h++

```
1  /*
2     In the GNU implementation of the C++ standard library (gcc
3     version 4.8.2), the algorithm underlying std::sort() was
4     introsort.
5
6     Source: David R. Musser, Introspective sorting and selection
7     algorithms, Software—Practice and Experience 27,8 (1997), 983–993
8
9     Author: Jyrki Katajainen © 2011
10
11    Worst–case running time: O(n lg n)
12    # branch mispredictions: 0.43 n lg n [observed]
13    Extra space: O(1) elements, O(lg n) words
14
15    Observed sorting time per n lg n [ns]; input: random permutation
16        1024      3.60
17       32768      3.59
18     1048576      3.51
19    33554432      3.50
20 */
```

```
21
22 #include <algorithm> // std::sort
```

introsort.h++

```
1  #include <algorithm> // std::partial_sort std::iter_swap
2  #include <cstddef> // std::ptrdiff_t
3  #include <iterator> // std::iterator_traits
4  #include <limits> // std::numeric_limits
5  #include <utility> // std::move
6
7  namespace introsort {
8
9    enum: std::ptrdiff_t {threshold = 16};
10
11   template<typename I, typename C>
12   void move_median_first(I a, I b, I c, C less) {
13     if (less(*a, *b))        {
14       if (less(*b, *c)) {
15         std::iter_swap(a, b);
16       }
17       else if (less(*a, *c)) {
18         std::iter_swap(a, c);
19       }
20     }
21     else if (less(*a, *c)) {
22       return;
23     }
24     else if (less(*b, *c)) {
25       std::iter_swap(a, c);
26     }
27     else {
28       std::iter_swap(a, b);
29     }
30   }
31
32   template<typename R, typename V, typename C>
33   R unguarded_partition(R p, R r, V const& v, C less) {
34     V x;
35     while (true) {
36       while (less(*p, v)) {
37         ++p;
38       }
39       --r;
40       while (less(v, *r)) {
41         --r;
42       }
43       if (p ≥ r) {
44         return p;
45       }
46       x = *p;
```

```
47        *p = *r;
48        *r = x;
49        ++p;
50      }
51    }
52
53    template<typename R, typename C>
54    inline R unguarded_partition_pivot(R first, R past, C less) {
55      R mid = first + (past - first) / 2;
56      move_median_first(first, mid, (past - 1), less);
57      return unguarded_partition(first + 1, past, *first, less);
58    }
59
60    template<typename R, typename N, typename C>
61    void introsort_loop(R first, R past, N depth_limit, C less) {
62      while (past - first > threshold) {
63        if (depth_limit == 0) {
64          std::partial_sort(first, past, past, less);
65          return;
66        }
67        --depth_limit;
68        R cut = unguarded_partition_pivot(first, past, less);
69        introsort_loop(cut, past, depth_limit, less);
70        past = cut;
71      }
72    }
73
74    template<typename R, typename C>
75    void unguarded_linear_insert(R past, C less) {
76      using V = typename std::iterator_traits<R>::value_type;
77      V val = std::move(*past);
78      R next = past;
79      --next;
80      while (less(val, *next)) {
81        *past = std::move(*next);
82        past = next;
83        --next;
84      }
85      *past = std::move(val);
86    }
87
88    template<typename R, typename C>
89    void insertion_sort(R first, R past, C less) {
90      using V = typename std::iterator_traits<R>::value_type;
91      if (first == past) {
92        return;
93      }
94      for (R i = first + 1; i != past; ++i) {
95        if (less(*i, *first)) {
96          V val = std::move(*i);
97          std::move_backward(first, i, i + 1);
98          *first = std::move(val);
```

```
99        }
100       else {
101         unguarded_linear_insert(i, less);
102       }
103     }
104   }
105
106   template<typename R, typename C>
107   inline void unguarded_insertion_sort(R first, R past, C less) {
108     for (R i = first; i != past; ++i) {
109       unguarded_linear_insert(i, less);
110     }
111   }
112
113   template<typename R, typename C>
114   void final_insertion_sort(R first, R past, C less) {
115     if (past - first > threshold) {
116       insertion_sort(first, first + threshold, less);
117       unguarded_insertion_sort(first + threshold, past, less);
118     }
119     else {
120       insertion_sort(first, past, less);
121     }
122   }
123
124   template<typename N>
125   inline N lg(N n) {
126     N k;
127     for (k = 0; n != 0; n>>=1) {
128       ++k;
129     }
130     return k - 1;
131   }
132
133   inline int lg(int n) {
134     using N = std::size_t;
135     constexpr N  char_bit = std::numeric_limits<unsigned
      ↪ char>::digits;
136     return sizeof(int) * char_bit  - 1 - __builtin_clz(n);
137   }
138
139   inline long lg(long n) {
140     using N = std::size_t;
141     constexpr N char_bit = std::numeric_limits<unsigned
      ↪ char>::digits;
142     return sizeof(long) * char_bit - 1 - __builtin_clzl(n);
143   }
144
145   inline long long lg(long long n) {
146     using N = std::size_t;
147     constexpr N char_bit = std::numeric_limits<unsigned
      ↪ char>::digits;
```

```
148       return sizeof(long long) * char_bit - 1 - __builtin_clzll(n);
149     }
150
151     template<typename R, typename C>
152     inline void sort(R first, R past, C less) {
153       if (first != past) {
154         introsort_loop(first, past, 2 * lg(past - first), less);
155         final_insertion_sort(first, past, less);
156       }
157     }
158   }
```

skewed_introsort.h++

```
1  /*
2     This quicksort program
3     - is half-recursive
4     - switches to heapsort if the recursion depth gets too high
5     - relies on skewed pivot selection
6     - uses Hoare's partitioning routine.
7
8     Warning: This is not a general-purpose sorting routine; it is
9     tuned for sorting permutations of integers {0,1,...,n-1}.
10
11    Source: Kanela Kaligosi and Peter Sanders, How branch
12    mispredictions affect quicksort, Proceedings of the 14th Annual
13    European Symposium on Algorithms, Lecture Notes in Computer
14    Science 4168, Springer-Verlag (2006), 780-791
15
16    Author: Jyrki Katajainen © 2011
17
18    Worst-case running time: O(n lg n)
19    Extra space: O(1) elements, O(lg n) words
20
21    Observed sorting time per n lg n [ns]; input: random permutation
22        1024      3.14
23       32768      3.02
24     1048576      2.93
25    33554432      2.91
26 */
27
28 #include <algorithm> // std::partial_sort std::move_backward
29 #include <cstddef> // std::ptrdiff_t
30 #include <iterator> // std::iterator_traits
31 #include <limits> // std::numeric_limits
32 #include <utility> // std::move
33
34 #ifndef ALPHA
35 #define ALPHA 5
36 #endif
37
```

```
38 namespace skewed_introsort {
39
40   enum: std::ptrdiff_t {threshold = 16};
41
42   template<typename V, typename R>
43   V pivot(R base, R p, R r) {
44     using D = typename std::iterator_traits<R>::difference_type;
45     D fraction = (D) ((1.0 / double(ALPHA)) * double(r - p));
46     R q = p + fraction;
47     return (V) (q - base);
48   }
49
50   template<typename R, typename V, typename C>
51   R  unguarded_partition(R p, R r, V const& v, C less) {
52     while (true) {
53       while (less(*p, v)) {
54         ++p;
55       }
56       --r;
57       while (less(v, *r)) {
58         --r;
59       }
60       if (p ≥ r) {
61         return p;
62       }
63       std::iter_swap(p++, r);
64     }
65   }
66
67   template<typename R, typename N, typename C>
68   void  introsort_loop(R base, R first, R past, N depth_limit, C
       ↪ less) {
69     using V = typename std::iterator_traits<R>::value_type;
70     while (past - first > threshold) {
71       if (depth_limit == 0) {
72         std::partial_sort(first, past, past, less);
73         return;
74       }
75       --depth_limit;
76       V v = pivot<V, R>(base, first, past);
77       R cut = unguarded_partition(first, past, v, less);
78       introsort_loop(base, cut, past, depth_limit, less);
79       past = cut;
80     }
81   }
82
83   template<typename R, typename C>
84   void unguarded_linear_insert(R past, C less) {
85     using V = typename std::iterator_traits<R>::value_type;
86     V val = std::move(*past);
87       R next = past;
88       --next;
```

```
89        while (less(val, *next)) {
90          *past = std::move(*next);
91          past = next;
92          --next;
93        }
94        *past = std::move(val);
95      }
96
97      template<typename R, typename C>
98      void insertion_sort(R first, R past, C less) {
99        using V = typename std::iterator_traits<R>::value_type;
100       if (first == past) {
101         return;
102       }
103       for (R i = first + 1; i != past; ++i) {
104         if (less(*i, *first)) {
105           V val = std::move(*i);
106           std::move_backward(first, i, i + 1);
107           *first = std::move(val);
108         }
109         else {
110           unguarded_linear_insert(i, less);
111         }
112       }
113     }
114
115     template<typename R, typename C>
116     inline void unguarded_insertion_sort(R first, R past, C less) {
117       for (R i = first; i != past; ++i) {
118         unguarded_linear_insert(i, less);
119       }
120     }
121
122     template<typename R, typename C>
123     void final_insertion_sort(R first, R past, C less) {
124       if (past - first > threshold) {
125         insertion_sort(first, first + threshold, less);
126         unguarded_insertion_sort(first + threshold, past, less);
127       }
128       else {
129         insertion_sort(first, past, less);
130       }
131     }
132
133     template<typename N>
134     inline N lg(N n) {
135       N k;
136       for (k = 0; n != 0; n>>=1) {
137         ++k;
138       }
139       return k - 1;
140     }
```

```
141
142   inline int lg(int n) {
143     using N = std::size_t;
144     constexpr N  char_bit = std::numeric_limits<unsigned
      ↪ char>::digits;
145     return sizeof(int) * char_bit  - 1 - __builtin_clz(n);
146   }
147
148   inline long lg(long n) {
149     using N = std::size_t;
150     constexpr N char_bit = std::numeric_limits<unsigned
      ↪ char>::digits;
151     return sizeof(long) * char_bit - 1 - __builtin_clzl(n);
152   }
153
154   inline long long lg(long long n) {
155     using N = std::size_t;
156     constexpr N char_bit = std::numeric_limits<unsigned
      ↪ char>::digits;
157     return sizeof(long long) * char_bit - 1 - __builtin_clzll(n);
158   }
159
160   template<typename R, typename C>
161   void sort(R first, R past, C less) {
162     if (first != past) {
163       introsort_loop(first, first, past, 2 * lg(past - first), less);
164       final_insertion_sort(first, past, less);
165     }
166   }
167 }
```

tuned_quicksort.h++

```
1  /*
2     This quicksort program
3     - is iterative
4     - relies on the median-of-three pivot selection
5     - uses a lean version of Lomuto's partitioning routine.
6
7     Warning: This is not a general-purpose sorting routine; it is
8     tuned for sorting permutations of integers {0,1,...,n-1}.
9
10    Source: Jyrki Katajainen, Branch mispredictions don't affect
11    mergesort, Slides used at the 11th International Symposium on
12    Experimental Algorithms, Bordeaux, 2012-06-08
13
14    Author: Jyrki Katajainen © 2012
15
16    Worst-case running time: O(n^2)
17    # branch mispredictions: O(n)
18    Extra space: O(1) elements, O(lg n) words
```

```
19
20      Observed sorting time per n lg n [ns]; input: random permutation
21          1024      2.76
22         32768      2.70
23       1048576      2.65
24      33554432      2.69
25  */
26
27  #include <algorithm> // std::__final_insertion_sort
28  #include <cstddef> // std::ptrdiff_t
29  #include "introsort.h++" // introsort::final_insertion_sort
30  #include <iterator> // std::iterator_traits
31
32  namespace tuned_quicksort {
33
34    enum: std::ptrdiff_t {threshold = 16};
35
36    template<typename R, typename C>
37    R pivot(R p, R r, C less) {
38      R last = r - 1;
39      R q = p + (r - p) / 2;
40      R t = less(*q, *p) ? p : q;
41      t = less(*t, *last) ? last : t;
42      R i = (t == p) ? q : p;
43      R j = (t == last) ? q : last;
44      j = less(*j, *i) ? i : j;
45      return j;
46    }
47
48    template<typename R, typename C>
49    R lean_partition(R p, R r, C less) {
50      using D = typename std::iterator_traits<R>::difference_type;
51      using V = typename std::iterator_traits<R>::value_type;
52      R q = pivot(p, r, less);
53      V const v = *q;
54      R const first = p;
55      *q = *first;
56      q = first + 1;
57      while (q < r) {
58        V x = *q;
59        bool smaller = less(x, v);
60        p += smaller;
61        D delta = smaller * (q - p);
62        R s = p + delta;
63        R t = q - delta;
64        *s = *p;
65        *t = x;
66        ++q;
67      }
68      *first = *p;
69      *p = v;
70      return p;
```

```
71    }
72
73    template<typename R, typename C>
74    void sort(R p, R r, C less) {
75      R stack[256];
76      R* s = stack;
77      *s = p;
78      *(s + 1) = r;
79      s += 2;
80      do {
81        if (r - p > threshold) {
82          R q = lean_partition(p, r, less);
83          if (q - p > r - q) {
84            *s = p;
85            *(s + 1) = q;
86            p = q + 1;
87          }
88          else {
89            *s = q + 1;
90            *(s + 1) = r;
91            r = q;
92          }
93          s += 2;
94        }
95        else {
96          s -= 2;
97          p = *s;
98          r = *(s + 1);
99        }
100     } while (s != stack);
101     introsort::final_insertion_sort(p, r, less);
102   }
103 }
```

median_for_free_quicksort.h++

```
1  /*
2     This quicksort program
3     - is iterative
4     - uses the midpoint of the current range as the pivot
5     - uses a lean version of Lomuto's partitioning routine.
6
7     Warning: This is not a general-purpose sorting routine; it is
8     tuned for sorting only permutations of integers {0,1,...,n-1}.
9
10    Source: Jyrki Katajainen, Branch mispredictions don't affect
11    mergesort, Slides used at the 11th International Symposium on
12    Experimental Algorithms, Bordeaux, 2012-06-08
13
14    Author: Jyrki Katajainen © 2012
15
```

```
16      Worst–case running time: ∞
17      Extra space: O(1) elements, O(lg n) words
18
19      Observed sorting time per n lg n [ns]; input: random permutation
20          1024      2.57
21         32768      2.46
22       1048576      2.33
23      33554432      2.31
24  */
25
26  #include <algorithm> // std::iter_swap
27  #include <cstddef> // std::ptrdiff_t
28  #include "introsort.h++" // introsort::final_insertion_sort
29  #include <iterator> // std::iterator_traits
30
31  namespace median_for_free_quicksort {
32
33    enum: std::ptrdiff_t {threshold = 16};
34
35    template<typename V, typename R>
36    V pivot(R base, R p, R r) {
37      using D = typename std::iterator_traits<R>::difference_type;
38      D fraction = (D) (0.5 * double(r - p));
39      R q = p + fraction;
40      return V(q - base);
41    }
42
43    template<typename R, typename C>
44    R lean_partition(R base, R p, R r, C less) {
45      using V = typename std::iterator_traits<R>::value_type;
46      using D = typename std::iterator_traits<R>::difference_type;
47      V v = pivot<V, R>(base, p, r);
48      R q = p;
49      while (q < r and ! less(*q, v)) {
50        ++q;
51      }
52      if (q == r) {
53        return p;
54      }
55      std::iter_swap(p, q);
56      ++q;
57      while (q < r) {
58        V x = *q;
59        bool smaller = less(x, v);
60        p += smaller;
61        D delta = smaller * (q - p);
62        R s = p + delta;
63        R t = q - delta;
64        *s = *p;
65        *t = x;
66        ++q;
67      }
```

```
68      return ++p;
69    }
70
71    template<typename R, typename C>
72    void sort(R p, R r, C less) {
73      R stack[256];
74      R const base = p;
75      R* s = stack;
76      *s = p;
77      *(s + 1) = r;
78      s += 2;
79      do {
80        if (r - p > threshold) {
81          R q = lean_partition(base, p, r, less);
82          if (q - p > r - q) {
83            *s = p;
84            *(s + 1) = q;
85            p = q;
86          }
87          else {
88            *s = q;
89            *(s + 1) = r;
90            r = q;
91          }
92          s += 2;
93        }
94        else {
95          s -= 2;
96          p = *s;
97          r = *(s + 1);
98        }
99      } while (s != stack);
100     introsort::final_insertion_sort(p, r, less);
101   }
102 }
```

§ 5 Scaffolding

timer.h++

```
1  /*
2    This high–resolution timer is able to measure the elapsed time with
3    one microsecond accuracy
4
5    Author: Song Ho Ahn (song.ahn@gmail.com) © 2003, 2006
6  */
7
8  #include <sys/time.h>
9
10 class timer {
```

```
11  public:
12
13    timer();
14    ~timer();
15
16    void start();
17    void stop();
18    double getElapsedTime();          // get elapsed time in seconds
19    double getElapsedTimeSec();       // same as getElapsedTime
20    double getElapsedTimeMilliSec();// get elapsed time in milliseconds
21    double getElapsedTimeMicroSec();// get elapsed time in microseconds
22    double getElapsedTimeNanoSec(); // get elapsed time in nanoseconds
23
24  private:
25
26    double startTimeMicroSec;         // starting time in microseconds
27    double endTimeMicroSec;           // ending time in microseconds
28    int stopped;                      // stop flag
29    timeval startCount;
30    timeval endCount;
31  };
32
33  #include "timer.i++"
```

timer.i++

```
1   /*
2     This high–resolution timer is able to measure the elapsed time with
3     one micro–second accuracy
4
5     Author: Song Ho Ahn (song.ahn@gmail.com) © 2003, 2006
6   */
7
8   #include <stdlib.h>
9
10  // constructor
11
12  timer::timer() {
13    startCount.tv_sec = startCount.tv_usec = 0;
14    endCount.tv_sec = endCount.tv_usec = 0;
15    stopped = 0;
16    startTimeMicroSec = 0;
17    endTimeMicroSec = 0;
18  }
19
20  // destructor
21
22  timer::~timer() {
23  }
24
25  // start timer; startCount will be set at this point
```

```
26
27 void timer::start() {
28   stopped = 0; // reset stop flag
29   gettimeofday(&startCount, nullptr);
30 }
31
32 // stop the timer; endCount will be set at this point
33
34 void timer::stop() {
35   stopped = 1; // set timer stopped flag
36   gettimeofday(&endCount, nullptr);
37 }
38
39 // multiply elapsedTimeMicroSec by 1000
40
41 double timer::getElapsedTimeNanoSec() {
42   return getElapsedTimeMicroSec() * 1000.0;
43 }
44
45 // compute elapsed time in micro-seconds
46 // other routines will call this, then convert to specific resolution
47
48 double timer::getElapsedTimeMicroSec() {
49   if (not stopped) {
50     gettimeofday(&endCount, nullptr);
51   }
52   startTimeMicroSec = (startCount.tv_sec * 1000000.0) +
       ↪ startCount.tv_usec;
53   endTimeMicroSec = (endCount.tv_sec * 1000000.0) +
       ↪ endCount.tv_usec;
54   return endTimeMicroSec - startTimeMicroSec;
55 }
56
57 // divide elapsedTimeMicroSec by 1000
58
59 double timer::getElapsedTimeMilliSec() {
60   return getElapsedTimeMicroSec() * 0.001;
61 }
62
63 // divide elapsedTimeMicroSec by 1000000
64
65 double timer::getElapsedTimeSec() {
66   return getElapsedTimeMicroSec() * 0.000001;
67 }
68
69 // same as getElapsedTimeSec()
70
71 double timer::getElapsedTime() {
72   return getElapsedTimeSec();
73 }
```

sort-driver.c++

```
#if not defined(MAXSIZE)
#define MAXSIZE (32*1024*1024)
#endif

#include <algorithm> // std::random_shuffle std::sort
#include <cmath> // ilogb
#include <functional> // std::less
#include <iostream>   // std::cout std::cerr
#include <iterator>   // std::iterator_traits
#include "timer.h++"

extern int ilogb(double) throw();

template<typename R>
void show(R a, R e) {
  while (a != e) {
    std::cout << int(*a) << " ";
    ++a;
  }
  std::cout << '\n';
}

template<typename R, typename D>
void show(R a, D n) {
  D i = 1;
  while (i <= n) {
    std::cout << int(a[i]) << " ";
    ++i;
  }
  std::cout << '\n';
}

template<typename R>
bool is_permutation(R first, R past) {
  using V = typename std::iterator_traits<R>::value_type;
  std::sort(first, past);
  for (R q = first; q != past; ++q) {
    V i = V(q - first);
    if (*q != i) {
      std::cerr << "n: " << past - first << '\n';
      std::cerr << i << " missing: " << *q << " instead\n";
      return false;
    }
  }
  return true;
}

template<typename R, typename C>
bool is_sorted(R first, R past, C less) {
```

```
50    using D = typename std::iterator_traits<R>::difference_type;
51    R const a = first - 1;
52    D const n = past - first;
53    bool violated = false;
54    for (D i = n; i > 1; i--) {
55      if (less(a[i], a[i - 1])) {
56        std::cerr << i << ": me " << a[i] << "; before "
57                  << a[i - 1] << '\n';
58        violated = true;
59      }
60    }
61    return not violated;
62  }
63
64  #include "algorithm.h++"
65  #include "do_nothing.h++"
66
67  #ifdef MEASURE_COMPARISONS
68
69  long long comparisons = 0;
70
71  template<typename T>
72  class counting_comparator {
73  public:
74
75    typedef T first_argument_type;
76    typedef T second_argument_type;
77    typedef bool result_type;
78
79    bool operator()(T const& a, T const& b) const {
80      ++comparisons;
81      return a < b;
82    }
83  };
84
85  #endif
86
87  #ifdef MEASURE_MOVES
88
89  long long moves = 0;
90
91  template<typename T>
92  class move_counter {
93  private:
94
95    T element;
96
97  public:
98
99    explicit move_counter()
100     : element(0) {
101   }
```

```
102
103   template<typename number>
104   explicit move_counter(number x = 0)
105     : element(x) {
106   }
107
108   move_counter(move_counter const& x) {
109     element = x;
110     moves += 1;
111   }
112
113   move_counter& operator=(move_counter const& other) {
114     element = other.element;
115     moves += 1;
116     return *this;
117   }
118
119   operator T() const {
120     return element;
121   }
122
123   template<typename U>
124   friend bool operator<(move_counter<U> const&,
        ↪ move_counter<U> const&);
125
126 };
127
128 template<typename T>
129 bool operator<(move_counter<T> const& x, move_counter<T>
        ↪ const& y) {
130   return x.element < y.element;
131 }
132
133 #endif
134
135 template<typename R>
136 void generate(R p, R r, char z) {
137   typedef typename std::iterator_traits<R>::value_type V;
138   switch (z) {
139   case 'd':
140     for (R q = p; q != r; ++q) {
141       *q = V((r - 1) - q);
142     }
143     break;
144   case 'i':
145     for (R q = p; q != r; ++q) {
146       *q = V(q - p);
147     }
148     break;
149   case 'r':
150     for (R q = p; q != r; ++q) {
151       *q = V(q - p);
```

```
152       }
153       std::random_shuffle(p, r);
154       break;
155     case 'z':
156       bool t = false;
157       for (R q = p; q != r; ++q) {
158         *q = V(t);
159         t = not t;
160       }
161       break;
162     }
163   }
164
165   void usage(int, char **argv) {
166     std::cerr << "Usage: " << argv[0]
167               << " <n> <'i'ncreasing | 'd'ecreasing | 'r'andom |
      ↪ 'b'ool>\n";
168     exit(1);
169   }
170
171   int main(int argc, char** argv) {
172
173   #ifdef MEASURE_MOVES
174
175     using V = move_counter<int>;
176
177   #else
178
179     using V = long long;
180
181   #endif
182
183   #ifdef MEASURE_COMPARISONS
184
185     using C = counting_comparator<V>;
186
187   #else
188
189     using C = std::less<V>;
190
191   #endif
192
193     unsigned int n;
194     char method;
195     if (argc == 1) {
196       n = 15;
197       method = 'i';
198     }
199     else if (argc == 2) {
200       n = atoi(argv[1]);
201       method = 'i';
202     }
```

```
203   else if (argc ≠ 3) {
204     usage(argc, argv);
205   }
206   else {
207     n = atoi(argv[1]);
208     method = *argv[2];
209   }
210   if (n < 1 or n > MAXSIZE) {
211     std::cerr << "n out of bounds [1.." << MAXSIZE << "]\n";
212     usage(argc, argv);
213   }
214   switch (method) {
215   case 'd':
216   case 'i':
217   case 'r':
218   case 'b':
219     break;
220   default:
221     std::cerr << "Method not in ['d','i','r','b']\n";
222     usage(argc, argv);
223   }
224
225   V* a = new V[MAXSIZE];
226   V* b = a;
227   for (volatile unsigned int t = MAXSIZE / n; t > 0; t--) {
228     generate(b, b + n, method);
229     b = b + n;
230   }
231
232   b = a;
233   timer clock;
234
235   clock.start();
236   for (volatile unsigned int t = MAXSIZE / n; t > 0; t--)  {
237     NAME::sort(b, b + n, C());
238     b = b + n;
239   }
240   clock.stop();
241
242 #if not defined(NDEBUG)
243
244   b = a;
245   for (volatile unsigned int t = MAXSIZE / n; t > 0; t--) {
246     bool ok = ::is_sorted(b, b + n, std::less<V>());
247     if (not ok) {
248       return 1;
249     }
250     if (method == 'd' or method == 'i' or method == 'r') {
251       ok = is_permutation(b, b + n);
252       if (not ok) {
253         show(b, b + n);
254         return 2;
```

```
255         }
256       }
257       b = b + n;
258     }
259
260   #endif
261
262     long long t = MAXSIZE / n;
263     t *= n * ilogb(n);
264
265   #ifdef MEASURE_COMPARISONS
266
267     std::cout.precision(3);
268     std::cout << n << '\t' << double(comparisons) / double(t) <<
        ↪ '\n';
269
270   #endif
271
272   #ifdef MEASURE_MOVES
273
274     std::cout.precision(3);
275     std::cout << n << '\t' << double(moves) / double(t) << '\n';
276
277   #endif
278
279   #if not defined(MEASURE_COMPARISONS) and not defined(MEASURE_MOVES)
280
281     double nano_seconds = clock.getElapsedTimeNanoSec();
282     std::cout.precision(3);
283     std::cout << n << '\t' << nano_seconds / double(t) << '\n';
284
285   #endif
286
287     delete[] a;
288     return 0;
289   }
```

makefile

```
1  CXX=g++
2  CXXFLAGS=-O3 -std=c++17 -Wall -Wextra -x c++ -DNDEBUG
3
4  header-files:= $(wildcard *.h++)
5  versions:= $(basename $(header-files))
6  time-tests:= $(addsuffix .time, $(versions))
7  comparison-tests:= $(addsuffix .comparisons, $(versions))
8  move-tests:= $(addsuffix .moves, $(versions))
9  misprediction-tests:= $(addsuffix .mispredictions, $(versions))
10 miss-tests:= $(addsuffix .misses, $(versions))
11
12 N = 1024 32768 1048576 33554432
13
14 $(time-tests): %.time : %.h++
15         @echo $* "time"
```

```
16          @cp $*.h++ algorithm.h++
17          $(CXX) $(CXXFLAGS) -DNAME=$* sort-driver.c++
18          @for n in $(N) ; do \
19            ./a.out $$n r ; \
20          done; \
21          rm -f ./a.out
22
23  $(comparison-tests): %.comparisons : %.h++
24          @echo $* "element comparisons"
25          @cp $*.h++ algorithm.h++
26          $(CXX) $(CXXFLAGS) -DNDEBUG -DMEASURE_COMPARISONS -DNAME=$*
        ↪ sort-driver.c++
27          @for n in $(N) ; do \
28            ./a.out $$n r ; \
29          done; \
30          rm -f ./a.out
31
32  $(move-tests): %.moves : %.h++
33          @echo $* "element moves"
34          @cp $*.h++ algorithm.h++
35          $(CXX) $(CXXFLAGS) -DNDEBUG -DMEASURE_MOVES -DNAME=$* sort-driver.c++
36          @for n in $(N) ; do \
37            ./a.out $$n r ; \
38          done; \
39          rm -f ./a.out
40
41  $(misprediction-tests): %.mispredictions : %.h++
42          @echo $* "branch-misprediction rate"
43          @cp $*.h++ algorithm.h++
44          @for n in $(N) ; do \
45            python branch_mispredictions.py $* $$n ; \
46            rm -f ./a.out ; \
47            rm -f ./cachegrind.out.* ; \
48          done
49
50  clean:
51          - rm  -f a.out *.o *.ii */*.o */*.ii cachegrind.out.* 2>/dev/null
52
53  veryclean: clean
54          - rm -f *~ */*~ 2>/dev/null
```

§ 6 **Conclusion**

Let us use the following shorthands for the theoretical properties of different sorting programs:

★ general purpose, i.e. suitable for a software library
× in situ (or in place)
○ $O(n \lg n)$ worst case where n is the input size
● $O(n)$ (or $O(1)$) branch mispredictions.

When sorting permutations, on Janus, the ranking order of the described sorting programs can now be summarized as follows. Each program is tagged with its theoretical properties.

1. median-for-free quicksort×●
2. tuned quicksort×●
3. skewed introsort×○
4. tuned mergesort★○●
5. `std::sort` ≡ introsort★×○

6. `std::stable_sort`★○●
7. median-for-free in-situ mergesort×○●
8. in-situ mergesort★×○●
9. lean mergesort★○●
10. Floyd's heapsort★×○
11. Williams' heapsort★×○
12. branch-optimized Williams' heapsort★×○●
13. lean heapsort★×○●
14. in-place `std::stable_sort`★×

We do dot consider quicksort and mergesort variants that are tuned for handling permutations efficiently as general-purpose sorting routines. In particular, Lomuto's partitioning is too sensitive when there are equal elements among the input, and pivot-selection strategies that calculate the pivot based on the current range are too sensitive to the distribution of the input values.

According to our experiments, tuned mergesort was the fastest general-purpose sorting routine, but it requires linear extra space. In-situ mergesort was the fastest general-purpose, in-situ sorting routine; still it is not stable. Be aware that this conclusion is drawn based on integer sorting. In more general case, the number of element comparisons and that of element moves can change the ranking of these programs.

It will be interesting to see what will be the relative ranking order of the described programs, say, in ten years from now (i.e. in 2024).