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.

Copyright notice

Copyright © 2000–2014 by The authors and Performance Engineering Laboratory (University of Copenhagen)

The programs included in the CPH STL are placed in the public domain. The files may be freely copied and distributed, provided that no changes whatsoever are made. Changes are permissible only if the modified files are given new names, different from the names of existing files in the CPH STL, and only if the modified files are clearly identified as not being part of the library. Usage of the source code in derived works is otherwise unrestricted.

The authors have tried to produce correct and useful programs, but no warranty of any kind should be assumed.

Release date

2014-12-11

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

`E`: 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, . . . ,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++

```
1  /*
2     This program is a translation of Floyd's original Algol program
3     into C++.
4
5     Source: Robert W. Floyd, Algorithm 245: Treesort 3, Communications
6     of the ACM 7,12 (1964), 701
7
8     Author: Jyrki Katajainen © 1999, 2011
9
10    Worst–case running time: O(n lg n)
11    # element comparisons: 2n 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      6.25
18       32768      6.21
19     1048576      7.22
20    33554432     12.9
21 */
22
23 #include <algorithm> // std::iter_swap
24 #include <iterator> // std::iterator_traits
25
26 namespace floyd {
27
28   template <typename R, typename index, typename C>
29   void sift_down(R a, index i, index n, C less) {
30     typedef typename std::iterator_traits<R>::value_type E;
31     index j;
32     E copy = a[i];
33   loop:
34     j = 2 * i;
35     if (j <= n) {
36       if (j < n) {
37         if (less(a[j], a[j + 1])) {
38           j = j + 1;
39         }
40       }
41       if (less(copy, a[j])) {
42         a[i] = a[j];
43         i = j;
44         goto loop;
45       }
46     }
47     a[i] = copy;
48   }
49
50   template <typename R, typename C>
51   void make_heap(R first, R past, C less) {
52     typedef typename std::iterator_traits<R>::difference_type index;
53     R const a = first - 1;
54     index const n = past - first;
55     for (index 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      typedef typename std::iterator_traits<R>::difference_type index;
63      R const a = first - 1;
64      index const n = past - first;
65      for (index i = n; i >= 2; --i) {
66        std::iter_swap(a + 1, a + i);
67        sift_down(a, index(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, C());
74      floyd::sort_heap(a, b, C());
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 of
6     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 index, typename C>
28    void sift_up(R a, index j, C less) {
29      typedef typename std::iterator_traits<R>::value_type E;
30      E in = a[j];
31    scan:
32      if (j > 1) {
33        index 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      typedef typename std::iterator_traits<R>::difference_type index;
46      index const n = past - first;
47      if (n < 2) {
48        return;
49      }
50      R const a = first - 1;
51      index 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 index, typename E, typename C>
59    void sift_down(R a, index n, E in, C less) {
60      index i = 1;
61    loop:
62      index 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      typedef typename std::iterator_traits<R>::difference_type index;
79      typedef typename std::iterator_traits<R>::value_type E;
80      index const n = past - first;
81      if (n < 2) {
82        return;
83      }
84      R const a = first - 1;
85      index i = n;
86    outheap:
87      E out = a[1];
88      E 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, C());
98      williams::sort_heap(a, b, C());
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 index, typename C>
36    void sift_up(R a, index j, C less) {
37      typedef typename std::iterator_traits<R>::value_type E;
38      E in = a[j];
39    scan:
40      if (j > 1) {
41        index 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      typedef typename std::iterator_traits<R>::difference_type index;
54      index const n = past - first;
55      if (n < 2) {
56        return;
57      }
58      R const a = first - 1;
59      index 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 index, typename E, typename C>
67    void sift_down(R a, index n, E in, C less) {
68      index i = 1;
69    loop:
70      index 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      typedef typename std::iterator_traits<R>::difference_type index;
85      typedef typename std::iterator_traits<R>::value_type E;
86      index const n = past - first;
87      if (n < 2) {
88        return;
89      }
90      R const a = first - 1;
91      index i = n;
92    outheap:
93      E out = a[1];
94      E 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, C());
104     optimized_williams::sort_heap(a, b, C());
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
```

```
#include <algorithm> // std::swap
#include <iterator> // std::iterator_traits

namespace lean_heapsort {

  template <typename R, typename index, typename C>
  void sift_up(R a, index j, C less) {
    typedef typename std::iterator_traits<R>::value_type E;
    E in = a[j];
    while (j > 1) {
      index i = j / 2;
      if (less(a[i], in)) {
        a[j] = a[i];
        j = i;
      }
      else {
        break;
      }
    }
    a[j] = in;
  }

  template <typename R, typename C>
  void make_heap(R first, R past, C less) {
    typedef typename std::iterator_traits<R>::difference_type index;
    typedef typename std::iterator_traits<R>::value_type E;
    R const a = first - 1;
    index const n = past - first;
    index const m = (n & 1) ? n : n - 1;
    index i = m / 2;
    index j = i;
    index hole = j;
    E copy;
    while (i > 0) {
      if (i == j) hole = j;
      if (i == j) copy = a[j];
      j = 2 * j;
      j += less(a[j], a[j + 1]);
      a[hole] = a[j];
      if (less(copy, a[j])) hole = j;
      if (2 * j > m) a[hole] = copy;
      if (2 * j > m) i = i - 1;
      if (2 * j > m) j = i;
    }
    sift_up(a, n, less);
  }

  template <typename R, typename C>
  void sort_heap(R first, R past, C less) {
    typedef typename std::iterator_traits<R>::difference_type index;
    typedef typename std::iterator_traits<R>::value_type E;
    index n = past - first;
    if (n < 2) {
      return;
    }
    R const a = first - 1;
    E out = a[1];
    E in = a[n];
    index j = 1;
    index hole = 1;
    while (n > 2) {
      j = 2 * j;
      j += less(a[j], a[j + 1]);
      a[hole] = a[j];
      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, C());
108     lean_heapsort::sort_heap(a, b, C());
109   }
110 }
```

§ 3 Mergesort

stable_sort.h++

```
1  /*
2     In the GNU implementation of the C++ standard library (gcc version
3     4.8.2), the algorithm underlying std::stable_sort() was bottom-up
4     mergesort.
5
6     Author: Jyrki Katajainen © 2011
7
8     Worst-case running time: O(n lg n)
9     # element comparisons: n lg n + O(n)
10    # element assignments: n lg n + O(n)
11    # branch mispredictions: O(n)
12    Extra space: n + O(1) elements, O(1) words
13
14    Observed sorting time per n lg n [ns]; input: random permutation
15        1024      3.54
16       32768      3.43
17     1048576      3.54
18    33554432      3.49
19 */
20
21 #include <algorithm> // std::stable_sort
22
23 namespace stable_sort {
24
25   template <typename R, typename C>
26   void sort(R a, R b, C less) {
27     std::stable_sort(a, b, less);
28   }
29 }
```

in_place_stable_sort.h++

```
1  /*
2     In the GNU implementation of the C++ standard library (gcc version
3     4.8.2), the algorithm underlying std::__inplace_stable_sort() was
4     an in-place variant of mergesort.
```

```
5
6     Author: Jyrki Katajainen © 2011
7
8     Worst–case running time: O(n (lg n)^2)
9     Extra space: O(1) elements, O(1) words
10
11    Observed sorting time per n lg n [ns]; input: random permutation
12        1024      17.0
13       32768      20.5
14     1048576      22.9
15    33554432      24.8
16 */
17
18 #include <algorithm> // std::__inplace_stable_sort
19
20 namespace in_place_stable_sort {
21
22   template <typename R, typename C>
23   void sort(R first, R past, C less) {
24     std::__inplace_stable_sort(first, past, less);
25   }
26 }
```

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 index, typename C>
34   void two_passes(input a, index n, C less) {
35     typedef typename std::iterator_traits<input>::value_type E;
36     index const rest = n & 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      E e1 = *u;
64      E e2 = *x;
65      E e3 = *v;
66      E 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 index, typename C>
76  input remaining_passes(input a, output b, index n, C less) {
77    index size = 4;
78    index 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) & 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 && 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     typedef typename std::iterator_traits<R>::value_type E;
139     typedef typename std::iterator_traits<R>::difference_type index;
140     index const n = past - first;
141     E* tmp = new E[n];
142     two_passes(first, n, less);
143     E* 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++

```
1  /*
2     This program implements bottom-up mergesort with the following two
3     optimizations:
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     - In the main loop of the merge routine, it moves four elements to
9       the output area in each iteration.
10
11    Source: Amr Elmasry, Jyrki Katajainen, and Max Stenmark, Branch
12    mispredictions don't affect mergesort, Proceedings of the 11th
13    International Symposium on Experimental Algorithms, Lecture Notes
14    in Computer Science 7276, Springer-Verlag (2012), 160-171
15
16    Author: Jyrki Katajainen © 2011
```

```
17
18    Worst-case running time: O(n lg n)
19    # element comparisons: n lg n + O(n)
20    # element assignments: n lg n + O(n)
21    # branch mispredictions: O(n)
22    Extra space: n + O(1) elements, O(1) words
23
24    Observed sorting time per n lg n [ns]; input: random permutation
25        1024     2.99
26       32768     3.05
27     1048576     3.18
28    33554432     3.27
29 */
30
31 #include <algorithm> // std::sort, std::copy
32 #include <iterator> // std::iterator_traits
33
34 namespace unrolled_mergesort {
35
36   template <typename input, typename index, typename C>
37   void two_passes(input a, index n, C less) {
38     typedef typename std::iterator_traits<input>::value_type E;
39     index const rest = n & 3;
40     input const boundary = a + n - rest;
41     for (input s = a; s < boundary; s += 4) {
42       input u = s;
43       input v = s + 1;
44       input x = s + 2;
45       input y = s + 3;
46       bool c = less(*v, *u);
47       input t = u;
48       u = c ? v : u;
49       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       E e1 = *u;
67       E e2 = *x;
68       E e3 = *v;
69       E 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 index, typename C>
79   input remaining_passes(input a, output b, index n, C less) {
80     index 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 && 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 && 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     typedef typename std::iterator_traits<R>::value_type E;
149     typedef typename std::iterator_traits<R>::difference_type index;
150     index const n = past - first;
151     E* tmp = new E[n];
152     two_passes(first, n, less);
153     E* 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     typedef typename C::second_argument_type first_argument_type;
39     typedef typename C::first_argument_type second_argument_type;
40     typedef bool result_type;
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 index, typename C>
53    void two_passes(input a, index n, C less) {
54      typedef typename std::iterator_traits<input>::value_type E;
55      index rest = n & 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        E e1 = *u;
83        E e2 = *x;
84        E e3 = *v;
85        E 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 index, typename C>
95    input remaining_passes(input a, output b, index n, C less) {
96      typedef typename std::iterator_traits<input>::value_type E;
97      index size = 4;
98      while (size < n) {
99        input p = a;
100       output o = b;
101       input r = a + n;
102       input t1 = nullptr;
103       while (p + size < r) {
104         t1 = p + size;
105         input t2 = (t1 + size < r) ? t1 + size : r;
106         input q = t1;
107         input t = nullptr;
108         E x;
109         while (p < t1 && q < t2) {
110           if (less(*q, *p)) {
111             t = q;
112             ++q;
113           }
```

```
114           else {
115             t = p;
116             ++p;
117           }
118           std::iter_swap(t, o++);
119         }
120         while (p < t1) {
121           std::iter_swap(p++, o++);
122         }
123         while (q < t2) {
124           std::iter_swap(q++, o++);
125         }
126         p = t2;
127       }
128       while (p < r) {
129         std::iter_swap(p++, o++);
130       }
131       size = size << 1;
132       std::swap(a, b);
133     }
134     return a;
135   }
136
137   template <typename R, typename C>
138   void mergesort(R p, R r, R t, C less) {
139     typedef typename std::iterator_traits<R>::difference_type index;
140     index const n = r - p;
141     two_passes(p, n, less);
142     R q = remaining_passes(p, t, n, less);
143     if (q != t) {
144       while (p != r) {
145         std::iter_swap(p++, t++);
146       }
147     }
148   }
149
150   template <typename R, typename C>
151   void sort(R p, R r, C less) {
152     typedef typename std::iterator_traits<R>::difference_type index;
153     index n = r - p;
154     index threshold = n / ilogb(2 + n);
155     while (n > threshold) {
156       R q_1 = p + n / 2;
157       R q_2 = r - n / 2;
158       converse_relation<C> greater(less);
159       std::nth_element(p, q_1, r, greater);
160       mergesort(p, q_1, q_2, less);
161       r = q_1;
162       n = r - p;
163     }
164     std::sort(p, r, less);
165   }
166 }
```

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 tuned
8    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, std::iter_swap
27 #include <cmath> // ilogb
28
29 namespace median_for_free_in_situ_mergesort {
30
31   template <typename C>
32   class converse_relation {
33   protected:
34     C less;
35
36   public:
37     typedef typename C::second_argument_type first_argument_type;
38     typedef typename C::first_argument_type second_argument_type;
39     typedef bool result_type;
40
41     explicit converse_relation(C const & less)
42       : less(less) {
43     }
44
45     bool operator()(typename C::first_argument_type const & x,
46                     typename C::second_argument_type const & y) const {
47       return less(y, x);
48     }
49   };
50
51   template <typename input, typename index, typename C>
52   void two_passes(input a, index n, C less) {
53     typedef typename std::iterator_traits<input>::value_type E;
54     index const rest = n & 3;
55     input const boundary = a + n - rest;
56     for (input s = a; s < boundary; s += 4) {
57       input u = s;
58       input v = s + 1;
59       input x = s + 2;
60       input y = s + 3;
61       bool c = less(*v, *u);
62       input t = u;
63       u = c ? v : u;
64       v = c ? t : v;
65       c = less(*y, *x);
66       t = x;
67       x = c ? y : x;
68       y = c ? t : y;
69       c = less(*x, *u);
70       t = u;
71       u = c ? x : u;
72       x = c ? t : x;
73       c = less(*y, *v);
```

```
74      t = v;
75      v = c ? y : v;
76      y = c ? t : y;
77      c = less(*v, *x);
78      t = x;
79      x = c ? v : x;
80      v = c ? t : v;
81      E e1 = *u;
82      E e2 = *x;
83      E e3 = *v;
84      E 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 index, typename C>
94  input remaining_passes(input a, output b, index n, C less) {
95    typedef typename std::iterator_traits<input>::value_type E;
96    index 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       E x;
108       while (p < t1 && q < t2) {
109         if (less(*q, *p)) {
110           t = q;
111           ++q;
112         }
113         else {
114           t = p;
115           ++p;
116         }
117         std::iter_swap(t, o++);
118       }
119       while (p < t1) {
120         std::iter_swap(p++, o++);
121       }
122       while (q < t2) {
123         std::iter_swap(q++, o++);
124       }
125       p = t2;
126     }
127     while (p < r) {
128       std::iter_swap(p++, o++);
129     }
130     size = size << 1;
131     std::swap(a, b);
132   }
133   return a;
134 }
135
136 template <typename R, typename C>
137 void mergesort(R p, R r, R t, C less) {
138   typedef typename std::iterator_traits<R>::difference_type index;
```

```
139     index const n = r - p;
140     two_passes(p, n, less);
141     R q = remaining_passes(p, t, n, less);
142     if (q != t) {
143       while (p != r) {
144         std::iter_swap(p++, t++);
145       }
146     }
147   }
148
149   template <typename R, typename E, typename C>
150   void partition(R p, R r, E v, C less) {
151     typedef typename std::iterator_traits<R>::difference_type index;
152     R q = p;
153     while (q < r && ! less(*q, v)) {
154       ++q;
155     }
156     if (q == r) {
157       return;
158     }
159     std::iter_swap(p, q);
160     ++q;
161     while (q < r) {
162       E x = *q;
163       bool smaller = less(x, v);
164       p += smaller;
165       index delta = smaller * (q - p);
166       R s = p + delta;
167       R t = q - delta;
168       *s = *p;
169       *t = x;
170       ++q;
171     }
172   }
173
174   template <typename R, typename C>
175   void sort(R p, R r, C less) {
176     typedef typename std::iterator_traits<R>::difference_type index;
177     typedef typename std::iterator_traits<R>::value_type E;
178     index n = r - p;
179     index threshold = n / (1 * ilogb(2 + n));
180     while (n > threshold) {
181       E median = (n - 1) / 2;
182       R q_1 = p + n / 2;
183       R q_2 = r - n / 2;
184       converse_relation<C> greater(less);
185       partition(p, r, median, greater);
186       mergesort(p, q_1, q_2, less);
187       r = q_1;
188       n = r - p;
189     }
190     std::sort(p, r, less);
191   }
192 }
```

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
27 #include <iterator> // std::iterator_traits
28
29 namespace lean_mergesort {
30
31   template <typename input, typename index, typename C>
32   void two_passes(input a, index n, C less) {
33     typedef typename std::iterator_traits<input>::value_type E;
34     index const rest = n & 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       E e1 = *u;
62       E e2 = *x;
63       E e3 = *v;
64       E 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 index, typename C>
74   input remaining_passes(input a, output b, index n, C less) {
75     index size = 4;
76     index i = 0;
77     index j = 0;
78     index k = 0;
79     index t1 = 0;
80     index t2 = 0;
81     input p = nullptr;
82     index 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    typedef typename std::iterator_traits<R>::value_type E;
109    typedef typename std::iterator_traits<R>::difference_type index;
110    index const n = past - first;
111    E* tmp = new E[n];
112    two_passes(first, n, less);
113    E* 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 version
3    4.8.2), the algorithm underlying std::sort() was introsort.
4
5    Source: David R. Musser, Introspective sorting and selection
6    algorithms, Software—Practice and Experience 27,8 (1997), 983–993
7
8    Author: Jyrki Katajainen © 2011
9
10   Worst–case running time: O(n lg n)
```

```
11    # branch mispredictions: 0.43 n lg n [observed]
12    Extra space: O(1) elements, O(lg n) words
13
14    Observed sorting time per n lg n [ns]; input: random permutation
15        1024      3.60
16       32768      3.59
17     1048576      3.51
18    33554432      3.50
19 */
20
21 #include <algorithm> // std::sort
```

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 tuned
9     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 Science
14    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::__final_insertion_sort, std::
       iter_swap, std::__lg
29 #include <cstddef> // std::size_t
30 #include <iterator> // std::iterator_traits
31
32 #ifndef ALPHA
33 #define ALPHA 5
34 #endif
35
36 namespace skewed_introsort {
37
38   enum: std::size_t {threshold = 16};
39
40   template <typename E, typename R>
41   E pivot(R base, R p, R r) {
42     typedef typename std::iterator_traits<R>::difference_type index;
43     index fraction = (index) ((1.0 / double(ALPHA)) * double(r - p));
44     R q = p + fraction;
45     return (E) (q - base);
46   }
47
48   template <typename R, typename E, typename C>
49   R  unguarded_partition(R p, R r, E const & v, C less) {
```

```
50      while (true) {
51        while (less(*p, v)) {
52          ++p;
53        }
54        --r;
55        while (less(v, *r)) {
56          --r;
57        }
58        if (p >= r) {
59          return p;
60        }
61        std::iter_swap(p++, r);
62      }
63    }
64
65    template <typename R, typename size, typename C>
66    void  introsort_loop(R base, R first, R past, size depth_limit, C less) {
67      typedef typename std::iterator_traits<R>::value_type E;
68      while (past - first > threshold) {
69        if (depth_limit == 0) {
70          std::partial_sort(first, past, past, less);
71          return;
72        }
73        --depth_limit;
74        E v = pivot<E, R>(base, first, past);
75        R cut = unguarded_partition(first, past, v, less);
76        introsort_loop(base, cut, past, depth_limit, less);
77        past = cut;
78      }
79    }
80
81    template <typename R, typename C>
82    void sort(R first, R past, C less) {
83      if (first != past) {
84        introsort_loop(first, first, past, std::__lg(past - first) * 2, less);
85        std::__final_insertion_sort(first, past, less);
86      }
87    }
88  }
```

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 tuned
8     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::size_t
29 #include <iterator> // std::iterator_traits
30
31 namespace tuned_quicksort {
32
33   enum: std::size_t {threshold = 16};
34
35   template <typename R, typename C>
36   R pivot(R p, R r, C less) {
37     R last = r - 1;
38     R q = p + (r - p) / 2;
39     R t = less(*q, *p) ? p : q;
40     t = less(*t, *last) ? last : t;
41     R i = (t == p) ? q : p;
42     R j = (t == last) ? q : last;
43     j = less(*j, *i) ? i : j;
44     return j;
45   }
46
47   template <typename R, typename C>
48   R lean_partition(R p, R r, C less) {
49     typedef typename std::iterator_traits<R>::difference_type index;
50     typedef typename std::iterator_traits<R>::value_type E;
51     R q = pivot(p, r, less);
52     E const v = *q;
53     R const first = p;
54     *q = *first;
55     q = first + 1;
56     while (q < r) {
57       E x = *q;
58       bool smaller = less(x, v);
59       p += smaller;
60       index delta = smaller * (q - p);
61       R s = p + delta;
62       R t = q - delta;
63       *s = *p;
64       *t = x;
65       ++q;
66     }
67     *first = *p;
68     *p = v;
69     return p;
70   }
71
72   template <typename R, typename C>
73   void sort(R p, R r, C less) {
74     R stack[256];
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(p, r, less);
82         if (q - p > r - q) {
83           *s = p;
84           *(s + 1) = q;
85           p = q + 1;
86         }
87         else {
```

```
88            *s = q + 1;
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     std::__final_insertion_sort(p, r, less);
101   }
102 }
```

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 tuned
8     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::__final_insertion_sort, std::iter_swap
27 #include <cstddef> // std::size_t
28 #include <iterator> // std::iterator_traits
29
30 namespace median_for_free_quicksort {
31
32   enum: std::size_t {threshold = 16};
33
34   template <typename E, typename R>
35   E pivot(R base, R p, R r) {
36     typedef typename std::iterator_traits<R>::difference_type index;
37     index fraction = (index) (0.5 * double(r - p));
38     R q = p + fraction;
39     return (E) (q - base);
40   }
41
42   template <typename R, typename C>
43   R lean_partition(R base, R p, R r, C less) {
44     typedef typename std::iterator_traits<R>::value_type E;
45     typedef typename std::iterator_traits<R>::difference_type index;
46     E v = pivot<E, R>(base, p, r);
```

```
47      R q = p;
48      while (q < r && ! less(*q, v)) {
49        ++q;
50      }
51      if (q == r) {
52        return p;
53      }
54      std::iter_swap(p, q);
55      ++q;
56      while (q < r) {
57        E x = *q;
58        bool smaller = less(x, v);
59        p += smaller;
60        index delta = smaller * (q - p);
61        R s = p + delta;
62        R t = q - delta;
63        *s = *p;
64        *t = x;
65        ++q;
66      }
67      return ++p;
68    }
69
70    template <typename R, typename C>
71    void sort(R p, R r, C less) {
72      R stack[256];
73      R const base = p;
74      R* s = stack;
75      *s = p;
76      *(s + 1) = r;
77      s += 2;
78      do {
79        if (r - p > threshold) {
80          R q = lean_partition(base, p, r, less);
81          if (q - p > r - q) {
82            *s = p;
83            *(s + 1) = q;
84            p = q;
85          }
86          else {
87            *s = q;
88            *(s + 1) = r;
89            r = q;
90          }
91          s += 2;
92        }
93        else {
94          s -= 2;
95          p = *s;
96          r = *(s + 1);
97        }
98      } while (s != stack);
99      std::__final_insertion_sort(p, r, less);
100   }
101 }
```

§ 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-second resolution
46 // other getElapsedTime will call this, then convert to correspond resolution
47
48 double timer::getElapsedTimeMicroSec() {
49   if (! 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++

```
1  #if ! defined(MAXSIZE)
2  #define MAXSIZE (64 * 1024 * 1024)
3  #endif
4
5  #if ! defined(NAME)
6  #define NAME std
7  #endif
8
9  #include <algorithm> // std::random_shuffle, std::sort
10 #include <cmath> // ilogb
11 #include <functional> // std::less
12 #include <iostream> // std::cout and std::cerr
13 #include <iterator> // std::iterator_traits
14 #include "timer.h++"
15
16 extern int ilogb(double) throw();
17
18 template <typename R>
```

```
19 bool is_permutation(R first, R past) {
20   typedef typename std::iterator_traits<R>::value_type E;
21   std::sort(first, past);
22   for (R q = first; q != past; ++q) {
23     E i = E(q - first);
24     if (*q != i) {
25       std::cerr << i << ": element missing " << *q << " instead" << std::endl;
26       return false;
27     }
28   }
29   return true;
30 }
31
32 template <typename R, typename C>
33 bool is_sorted(R first, R past, C less) {
34   typedef typename std::iterator_traits<R>::difference_type index;
35   const R a = first - 1;
36   const index n = past - first;
37   bool violated = false;
38   for (index i = n; i > 1; i--)
39     if (less(a[i], a[i - 1])) {
40       std::cerr << i << ": me " << a[i] << "; before " << a[i - 1] << std::endl;
41       violated = true;
42     }
43   return ! violated;
44 }
45
46 #include "algorithm.h++"
47 #include "do_nothing.h++"
48
49 #ifdef MEASURE_COMPARISONS
50
51 long volatile comparisons = 0;
52
53 template <typename E>
54 class counting_comparator {
55 public:
56
57   typedef E first_argument_type;
58   typedef E second_argument_type;
59   typedef bool result_type;
60
61   bool operator()(E const & a, E const & b) const {
62     ++comparisons;
63     return a < b;
64   }
65 };
66
67 #endif
68
69 #ifdef MEASURE_ASSIGNMENTS
70
71 long volatile assignments = 0;
72
73 template <typename E>
74 class move_counter {
75 private:
76
77   E element;
78
79 public:
80
81   explicit move_counter()
82     : element(0) {
83   }
```

```
84
85    template <typename number>
86    explicit move_counter(number x = 0)
87      : element(x) {
88    }
89
90    move_counter(move_counter const & x) {
91      element = x;
92      assignments += 1;
93    }
94
95    move_counter & operator=(move_counter const & other) {
96      element = other.element;
97      assignments += 1;
98      return *this;
99    }
100
101   operator E() const {
102     return element;
103   }
104
105   template <typename F>
106   friend bool operator<(move_counter<F> const &, move_counter<F> const &);
107
108   template <typename F>
109   friend bool operator==(move_counter<F> const &, move_counter<F> const &);
110
111 };
112
113 template <typename E>
114 bool operator<(move_counter<E> const & x, move_counter<E> const & y) {
115   return x.element < y.element;
116 }
117
118 template <typename E>
119 bool operator==(move_counter<E> const & x, move_counter<E> const & y) {
120   return x.element == y.element;
121 }
122
123 #endif
124
125 template <typename R>
126 void generate(R p, R r, char z) {
127   typedef typename std::iterator_traits<R>::value_type E;
128   switch (z) {
129   case 'd':
130     for (R q = p; q < r; ++q) {
131       *q = E((r - 1) - q);
132     }
133     break;
134   case 'i':
135     for (R q = p; q < r; ++q) {
136       *q = E(q - p);
137     }
138     break;
139   case 'r':
140     for (R q = p; q < r; ++q) {
141       *q = E(q - p);
142     }
143     std::random_shuffle(p, r);
144     break;
145   case 'z':
146     bool t = false;
147     for (R q = p; q < r; ++q) {
148       *q = E(t);
```

```
149       t = ! t;
150     }
151     break;
152   }
153 }
154
155 void usage(int argc, char **argv) {
156   std::cerr << "Usage: " << argv[0]
157             << " <N> <'i'ncreasing | 'd'ecreasing | 'r'andom | 'b'ool>"
158             << std::endl;
159   exit(1);
160 }
161
162 int main(int argc, char** argv) {
163
164 #ifdef MEASURE_ASSIGNMENTS
165
166   typedef move_counter<int> E;
167
168 #else
169
170   typedef int E;
171
172 #endif
173
174 #ifdef MEASURE_COMPARISONS
175
176   typedef counting_comparator<E> C;
177
178 #else
179
180   typedef std::less<E> C;
181
182 #endif
183
184   unsigned int N;
185   char method;
186   if (argc == 1) {
187     N = 15;
188     method = 'i';
189   }
190   else if (argc == 2) {
191     N = atoi(argv[1]);
192     method = 'i';
193   }
194   else if (argc != 3) {
195     usage(argc, argv);
196   }
197   else {
198     N = atoi(argv[1]);
199     method = *argv[2];
200   }
201   if (N < 1 || N > MAXSIZE) {
202     std::cerr << "N out of bounds [1.."
203               << MAXSIZE
204               << "]"
205               << std::endl;
206     usage(argc, argv);
207   };
208   switch (method) {
209   case 'd':
210   case 'i':
211   case 'r':
212   case 'b':
213     break;
```

```
214   default:
215     std::cerr << "Method not in ['d','i','r','b']" << std::endl;
216     usage(argc, argv);
217   }
218
219   E* a = new E[MAXSIZE];
220   E* b = a;
221   for (volatile unsigned int t = MAXSIZE / N; t > 0; t--) {
222     generate(b, b + N, method);
223     b = b + N;
224   }
225
226 #if defined(MEASURE_ASSIGNMENTS)
227
228   assignments = 0;
229
230 #elif defined(MEASURE_COMPARISONS)
231
232   comparisons = 0;
233
234 #endif
235
236   b = a;
237   timer clock;
238   clock.start();
239   for (volatile unsigned int t = 0; t < MAXSIZE / N; ++t)  {
240     NAME::sort(b, b + N, C());
241     b = b + N;
242   }
243   clock.stop();
244
245 #if ! defined(NDEBUG)
246
247   b = a;
248   for (volatile unsigned int t = 0; t < MAXSIZE / N; ++t) {
249     bool ok = ::is_sorted(b, b + N, std::less<E>());
250     if (! ok) {
251       return 1;
252     }
253     if (method == 'd' || method == 'i' || method == 'r') {
254       ok = ::is_permutation(b, b + N);
255       if (! ok) {
256         return 2;
257       }
258     }
259     b = b + N;
260   }
261
262 #endif
263
264   unsigned int t = (MAXSIZE / N) * N * ilogb(N);
265
266 #if defined(MEASURE_COMPARISONS)
267
268   std::cout.precision(3);
269   std::cout << N << '\t' << double(comparisons) / double(t) << std::endl;
270
271 #elif defined(MEASURE_ASSIGNMENTS)
272
273   std::cout.precision(3);
274   std::cout << N << '\t' << double(assignments) / double(t) << std::endl;
275
276 #else
277
278   double nano_seconds = clock.getElapsedTimeNanoSec();
```

```
279   std::cout.precision(3);
280   std::cout << N << '\t' << nano_seconds / double(t) << std::endl;
281
282 #endif
283
284   delete[] a;
285   return 0;
286 }
```

makefile

```
 1 CXX=g++
 2 CXXFLAGS=-O3 -Wall -std=c++11
 3 TESTFLAGS=-O3 -Wall -pedantic -x c++ -std=c++11 -g
 4
 5 header-files:= $(wildcard *.h++)
 6 versions:= $(basename $(header-files))
 7 time-tests:= $(addsuffix .time, $(versions))
 8 comparison-tests:= $(addsuffix .comparisons, $(versions))
 9 assignment-tests:= $(addsuffix .assignments, $(versions))
10 misprediction-tests:= $(addsuffix .mispredictions, $(versions))
11 miss-tests:= $(addsuffix .misses, $(versions))
12
13 N = 1024 32768 1048576 33554432
14
15 $(time-tests): %.time : %.h++
16         @echo $* "time"
17         @cp $*.h++ algorithm.h++
18         $(CXX) $(CXXFLAGS) -DNAME=$* sort-driver.c++
19         @for n in $(N) ; do \
20           ./a.out $$n r ; \
21         done; \
22         rm -f ./a.out
23
24 $(comparison-tests): %.comparisons : %.h++
25         @echo $* "element comparisons"
26         @cp $*.h++ algorithm.h++
27         $(CXX) $(CXXFLAGS) -DNDEBUG -DMEASURE_COMPARISONS -DNAME=$* sort-driver.c
               ++
28         @for n in $(N) ; do \
29           ./a.out $$n r ; \
30         done; \
31         rm -f ./a.out
32
33 $(assignment-tests): %.assignments : %.h++
34         @echo $* "element assignments"
35         @cp $*.h++ algorithm.h++
36         $(CXX) $(CXXFLAGS) -DNDEBUG -DMEASURE_ASSIGNMENTS -DNAME=$* sort-driver.c
               ++
37         @for n in $(N) ; do \
38           ./a.out $$n r ; \
39         done; \
40         rm -f ./a.out
41
42 $(misprediction-tests): %.mispredictions : %.h++
43         @echo $* "branch-misprediction rate"
44         @cp $*.h++ algorithm.h++
45         @for n in $(N) ; do \
46           python branch_mispredictions.py $* $$n ; \
47           rm -f ./a.out ; \
48           rm -f ./cachegrind.out.* ; \
49         done
50
51 clean:
```

```
52          - rm  -f a.out *.o *.ii */*.o */*.ii cachegrind.out.* 2>/dev/null
53
54 veryclean: clean
55          - 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).