

Performance Engineering the `nth_element` Function

Lars Yde

February 25, 2002

Contents

1	Introduction	4
2	The <code>nth_element</code> function	4
3	Purpose and objectives	4
4	Overview	5
5	<code>trivial_nth_element</code>	5
6	<code>select</code>	7
7	<code>randized_select</code>	9
8	<code>lazy_select</code>	13
9	Choosing an implementation	17
10	Conclusion	17
A	Source code	21

1 Introduction

2 The `nth_element` function

The `nth_element` function is part of the Standard Template Library which is described in the standards document for the C++ programming language [1]. Its purpose is to solve the selection problem[5, p.187] for a finite range of elements which is passed to the function as a pair of iterators, one to the beginning of the range and one to its end. In addition, the function is passed an iterator to the n th position of the range, where n is less than the length of the range. The range is then partitioned about the element that would occupy the n th position if the range were sorted.

Put differently, the prototype of the `nth_element` function can be written as follows using C++ code:

```
/* Preconditions: [first, nth) is a valid range.
   [nth, last) is a valid range.
   Postconditions: for all iterators i in [first, nth), !(*nth < *i)
                   for all iterators j in [nth, last), !(*j < *nth)
*/
template< class Iterator >
void nth_element( Iterator first, Iterator nth, Iterator last );
```

Furthermore, the `nth_element` is overloaded with a version that accepts an extra argument: a comparison function or function object which specifies an ordering on the elements in the range `[first, last)`. This comparison function is then used instead of `operator<` when elements in the range are compared to the n th element.

3 Purpose and objectives

The purpose of this report has been to evaluate different implementations of the `nth_element` function in order to determine which is best suited for inclusion in the Copenhagen STL [10]. The criteria on which such an assessment is based are correctness, efficiency and flexibility, in that order of significance. In other words, the implementation chosen must:

- Comply with the C++ standard.¹
- Have an asymptotic complexity better than or equal to that of existing implementations.
- Have lower running times than existing implementations of `nth_element` when passed identical inputs.

¹This may seem self-evident, but as Bentley notes in [2], even veteran programmers make mistakes when implementing seemingly simple algorithms.

- Meet and preferably surpass the iterator requirements of existing `nth_element` implementations. By surpassing, I mean allowing other types of iterator than existing implementations which all support the random access iterator type mandated by the C++ standard.

The last item means that the implementation decided upon should work with as many types of iterator as possible, not simply the random access iterator type specified by the C++ standard.

The focus in this report is on code, not algorithms. The below descriptions of the implementations evaluated focuses on the how's and why's of the code itself and gives only a summary description of the algorithms they are based upon — further information can be found in the relevant literature.

4 Overview

The implementations tested were chosen because they cover the range of possible approaches to solving the selection problem quite well.

`trivial_nth_element` simply delegates the problem to a third-party component, in this case a compiler-specific implementation of the STL `nth_element` function.

`randized_select` is an application of the divide-and-conquer paradigm using a random element as the “divider”, i.e. pivot element, with each recursive step.

`select` is another divide-and-conquer implementation, but uses a deterministically chosen pivot element when it divides the input data, namely a median-of-medians.

`lazy_select` is a randomized implementation that relies on a sampling technique to find the `nth` element in a constant number of steps with high probability.

The implementations are described below, and are listed with two types of benchmarking results: one which benchmarks running time, and another which benchmarks the number of **visits** made. By **visit**, I mean the execution of an instruction that will potentially result in a CPU reference to primary memory². This latter benchmark is made to give a more precise measurement of the (machine independent) time complexity of the implementation than the standard asymptotic O-analysis yields.

5 `trivial_nth_element`

The C++ standard stipulates that a compliant implementation of the `nth_element` function must accept random access iterators³ [1], and thus does not as such require that other types of iterator be supported. It does, however, not disallow support for such types and as mentioned earlier, I wanted the `cphestl::nth_element` implementation to be as “lax” as possible in its iterator requirements since, obviously, this could only extend the potential usefulness of the function.

²The idea of benchmarking visits derives from work made by Jyrki Katajainen.

³Iterators that support integer displacement as in `I + n` where `I` is an iterator and `n` an integer.

A trivial approach to supporting more than random access iterators is the following. Assume that `first` and `last` are iterators designating a range `[first, last)`, and that `nth` designates the n th element. Assume also that `first`, `last`, `nth` are all of type T , where T is not random access.

1. Copy `[first, last)` to a vector V .
2. Determine the distance d between `first` and `nth`.
3. Pass V and $V.begin() + d$ to an existing `nth_element` implementation, i.e. one that accepts only random access iterators.
4. Copy V to `[first, last)`.
5. Return `[first, last)` to the caller of the function.

The `nth_element` function used can be any of the implementations available, including of course the other implementations discussed in this report. Since this method circumvents the iterator requirements of whatever `nth_element` implementation it is used with, it is not limited by them. Instead, its iterator requirements is determined by the calls to `copy` which requires input iterators as its first two arguments and an output iterator as its third. Since iterator `first` must be used as both a first argument in a call to `copy` (when `[first, last)` is copied to V) and as a third argument (when V is copied to `[first, last)`), it must be both an input and an output iterator and therefore of type forward iterator or better. Iterators `nth` and `last` must be of the same type since any standard compliant `nth_element` implementation must have only one template parameter. Ergo, the trivial method, if implemented as described above, will work with forward, bidirectional or random access iterators.

The drawback of the trivial approach is the extra storage required by the intermediate container V . The redeeming feature, however, is its simplicity and thus ease of implementation and maintenance. To determine how this method fared in practice, I implemented two separate versions of it: one that mirrored as closely as possible the pseudo-code description above, and another that incorporated a small optimization, but was otherwise identical to the first. Both implementations are shown in appendix A as `trivial_nth_element_plain` and `trivial_nth_element_optimized`, respectively.

The two implementations both adhere closely to the simple pseudo-code description above, and differ only in how they implement the first two steps of it. Function `trivial_nth_element_plain` traverses the range `[first, last)` and for each iterator `iter` in that range, it copies `*iter` to its vector variable v and tests for `iter==nth`. When `iter` becomes equal to `nth`, the distance between `first` and `nth` is assigned to a variable storing d . If we assume that on average `iter == nth` at position `first + (last - first) / 2`, the number of `iter == nth` operations carried out by function `trivial_nth_element_plain` is twice as great as needed to determine d . I therefore wrote `trivial_nth_element_optimized` which uses two loops for copying elements from `[first, last)` to a vector v . Each iteration through the first loop copies an element and tests for `iter==nth`, exactly as in the first function. However, when the test is successful, execution skips the first loop and continues in the second loop which copies the remaining elements (the range `(nth, last)`) to v . This second loop does not test for `iter==nth`, however, and therefore does away with the redundant comparison operations carried out by `trivial_nth_element_plain`.

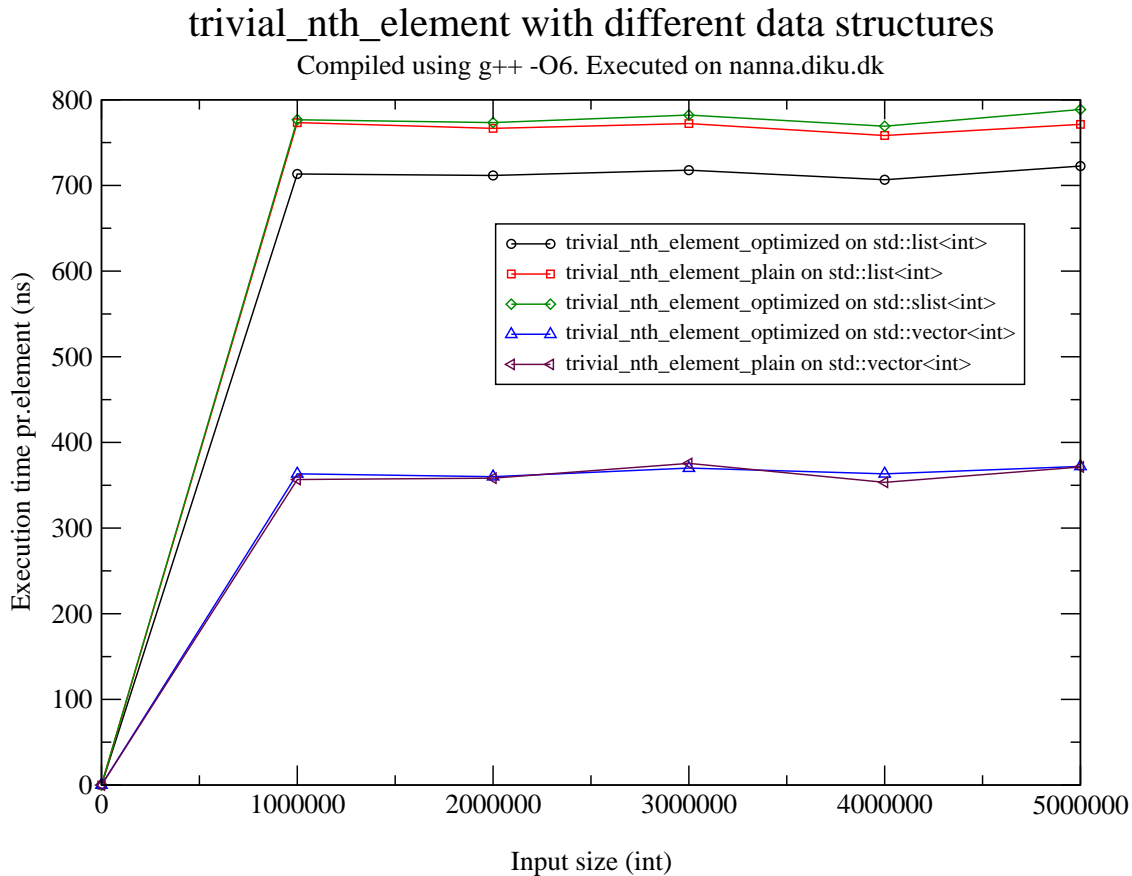


Figure 1: `trivial_nth_element` execution time with input ranges of different sizes.

Figure 1 shows the relative performance of the two `trivial_nth_element` implementations⁴ and indicates that the optimized version is 2-5% faster on average than its plain counterpart, which is hardly statistically significant. This is what could be expected since the number of operations eliminated is linear⁵ (thus accounting for the constant factor reduction in run-time), but the cost of each individual operation is very small (accounting for the modest savings overall).

6 select

The `select` implementation is based on code originally written by Jyrki Katajainen. I modified the code to work with forward and bidirectional iterators, and varied certain elements of the code in order to optimize performance. These efforts resulted in three different versions of `select`, all listed in appendix A.

The `select` implementation is based on the PICK algorithm by Blum et al. [3]. Briefly put, PICK operates by recursively partitioning an input range around a pivot element `p` which is

⁴Random access iterator benchmarking is included for completeness, although it is of little practical interest for obvious reasons.

⁵assuming a uniform distribution of input data.

Trivial_nth_element with different input sizes.
Compiled using gcc -O6. Executed on nanna.diku.dk

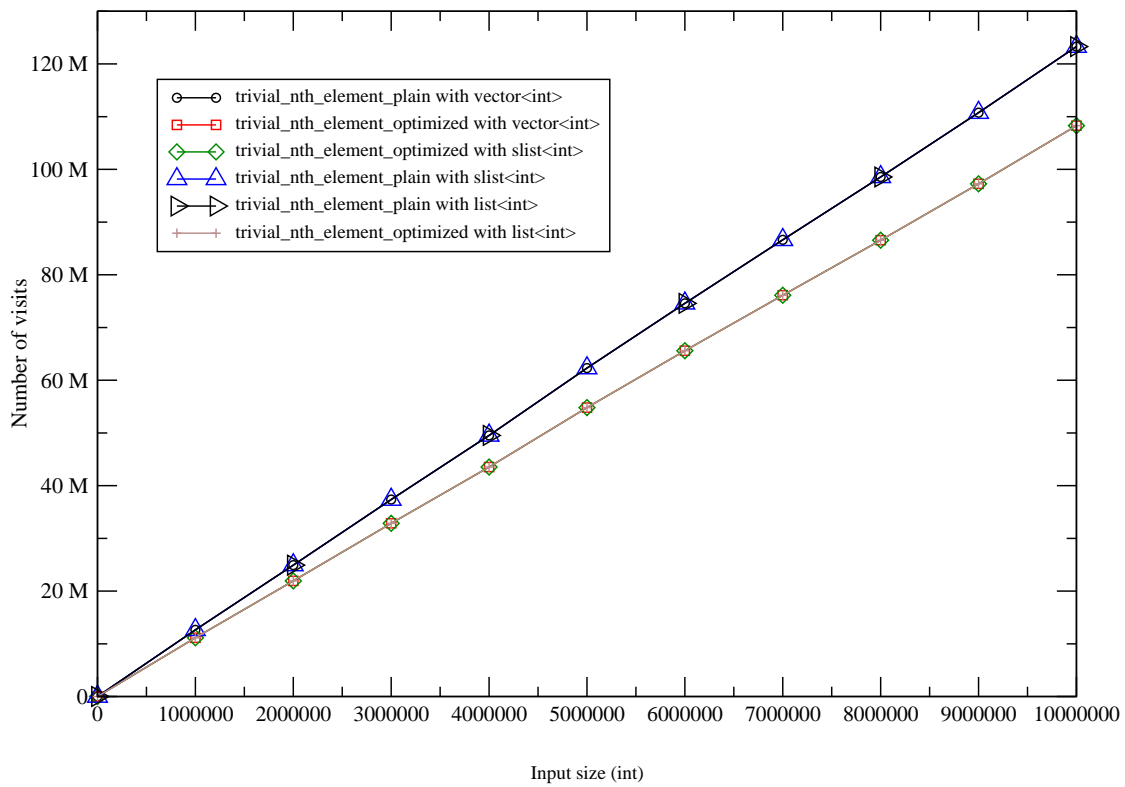


Figure 2: Number of visits while executing `trivial_nth_element`. Notice the lower number of visits made by the “loop-skipping” `trivial_nth_element_optimized` version.

found by finding the medians of m equally sized, disjoint subsets of the input range, and then finding the median of those medians. In other words, PICK guarantees a good split by using a median-of-medians as the pivot element in each recursive step.

The fact that PICK guarantees a good split makes it possible to reason exactly about its complexity since there are known bounds of the minimum and maximum number of elements that can be eliminated with each recursive step. It also means that it is difficult to improve on the basic design of PICK since the number of recursive steps (and hence visits) is dependent on the quality of the pivot element, and the median-of-medians seems to be the best possible choice of deterministic pivot. The partitioning step is equally difficult to improve on — after all, to partition a range, each element must be visited at least once.

I therefore focused my optimization efforts on specific elements of the `select` implementation rather than try to revamp the basic design. Three versions resulted:

- `select_with_pq_distance_optimization`
- `select_without_pq_distance_optimization`
- `select_with_unrolled_median`

All three versions were benchmarked — the results are shown in figure 3. As is clearly evident, `select_with_unrolled_median` is markedly superior to the other two versions. Averaged over the three types of data structure used and the different sizes of input, it is approximately 30% faster, making it a good contender for inclusion into the Copenhagen STL since it is now an asymptotically linear implementation with good runtime performance, and with modest iterator requirements, i.e. supporting forward iterators and up.

7 `randized_select`

The `randized_select` function was originally implemented by Jyrki Katajainen based on the classic FIND algorithm by C.A.R. Hoare [6]. Contrary to PICK — the algorithm that `select` is based on — FIND does not guarantee a good split; in fact, it is characterized by the complete absence of such a guarantee since the pivot element around which it partitions its input sequence is selected at random. In theory, this implies a worst case complexity of $O(n^2)$, but in practice, `randized_select` performs quite well as can be seen from the below benchmarking figures 4 and 5.

There is only one version of the `randized_select` implementation since the scope for optimization is quite limited, given the simplicity of the underlying design. The partitioning itself (carried out in function `partition_rselect`) carries a slight overhead compared to the original version in Katajainens implementation, and might appear a bit convoluted. The reason is that it must function with forward, bidirectional and random access iterators, which disallows some operations. This further limits the scope for optimization since such broad iterator support can conflict with optimizations that may rely on being able to randomly access iterators.

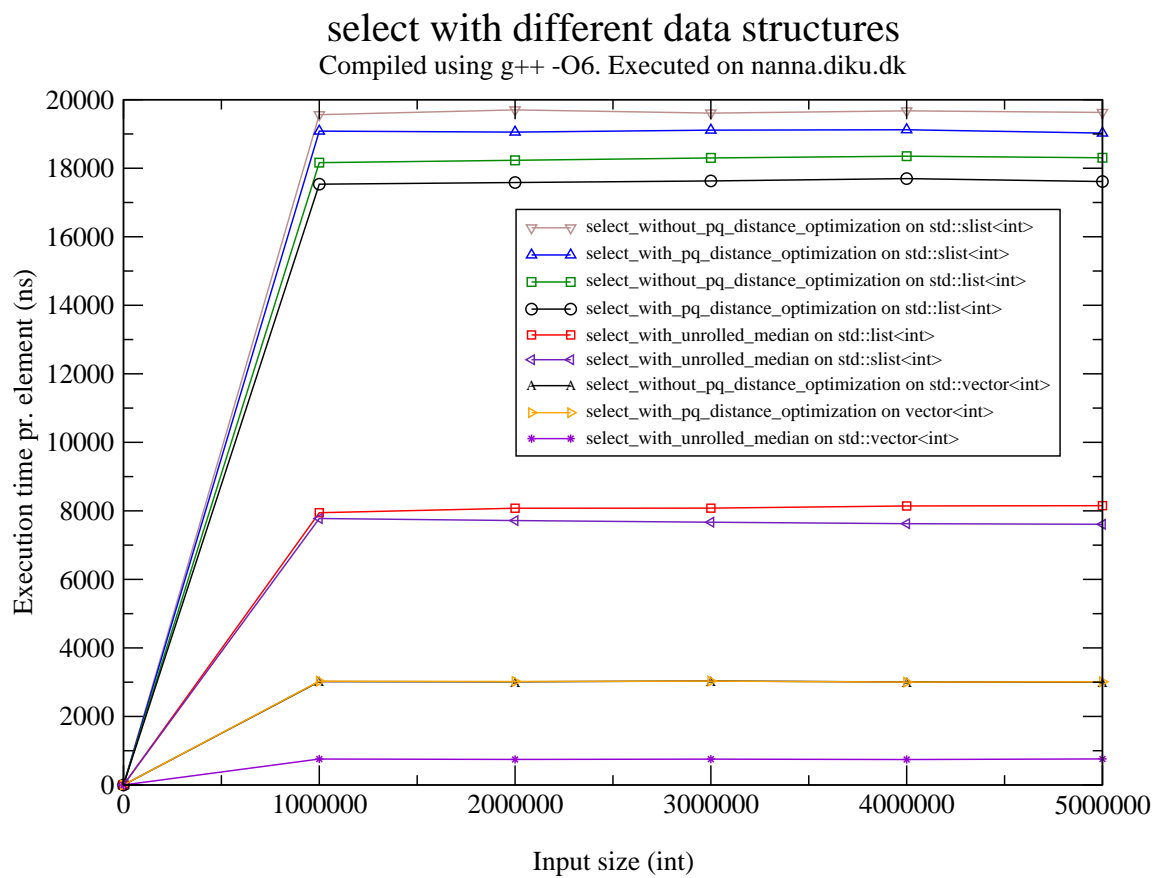


Figure 3: `select` execution time with input ranges of different sizes.

randized_select with different data structures

Compiled using g++ -O6. Executed on nanna.diku.dk

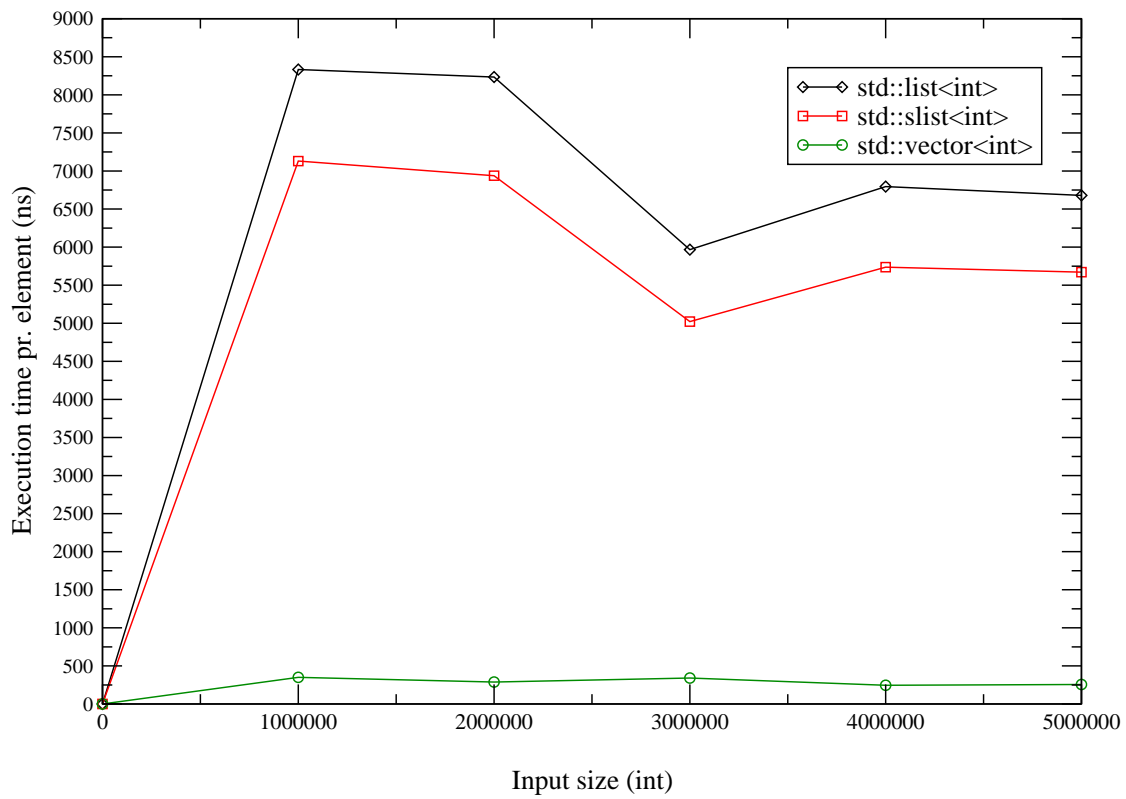


Figure 4: `randized_select` execution time with input ranges of different sizes.

randized_select with different data structures

Compiled using g++ -O6. Executed on nanna.diku.dk

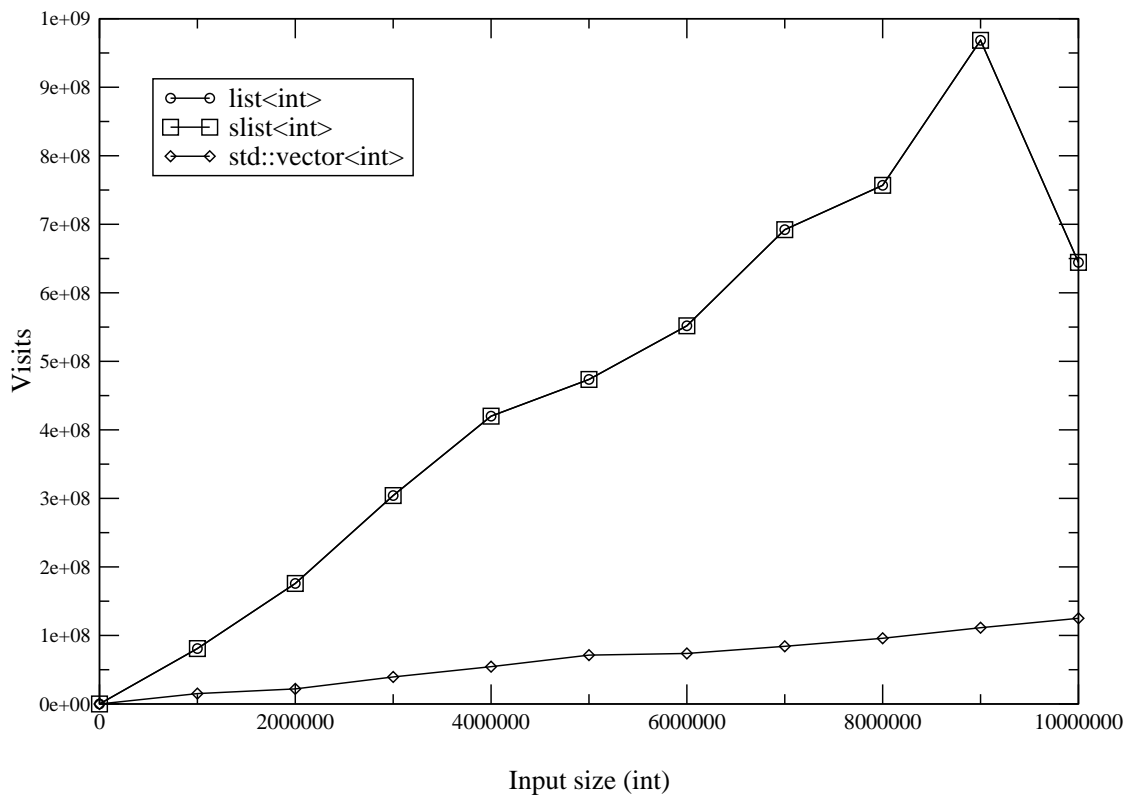


Figure 5: Memory visits made during execution of `randized_select`.

8 lazy_select

The two deterministic algorithms underlying `select` and `randized_select`, namely PICK and FIND are recursive algorithms that differ only in the method by which they select their pivot elements, and the consequent guarantees they make about the quality of those pivots. Selection through random sampling is different from both of these techniques. The idea is to draw a sample S at random (but uniformly) from the input range R , to find a pair x, y of elements from S , and then partition the entire input range using x, y . The partitioned range P is thus divided into three sections:

$$\boxed{\{e : e \in R \wedge e < x\} \mid \{e : e \in R \wedge x \leq e \leq y\} \mid \{e : e \in R \wedge e > y\}}$$

If the n th element is in the middle section of the partitioned range, it can be moved to the desired position by sorting the middle section, or, as in this case, using a standard, deterministic selection routine to partition the middle section around the n th element. If the n th element is in either of the two end sections, use the selection routine to partition it. Of course, the usefulness of the random sampling algorithm depends on two things:

1. The element we seek is in the middle section of the partitioned range.
2. The middle section is small enough to be partitioned inexpensively.

If either of these is not true, the cost of executing the algorithm will be dominated by the cost of the selection routine used, making it simply a costly alternative to using that routine by itself. The probability of partitioning well, i.e. in a manner that will place the n th element in the middle section and make that section comfortably small, is dependent on the choice of x, y values.

The random sampling algorithm is originally credited to Floyd-Rivest [8], but the version implemented here (listed as `lazy_select` in the appendix) is based on the exposition in [8], on [7], on [9], and in some part on an implementation of a similar algorithm by Jesper Bojesen [4]. The choice of x, y values differ from one work to the next. For example, in [8], the recommendation is to select x at position $n(|R|)^{\frac{1}{4}} - \sqrt{|R|}$, where R is the input range, or at position 1, whichever is larger. The y value is selected in a symmetrical fashion from the opposite “end” of the input range. In [9], Sibeyn gives an alternative choice of constants, but fundamentally, his selection scheme is also that of using proportional constants to index the sample S in order to find promising x, y values.

Function `lazy_select` uses the basic design by Sibeyn [9], and therefore also his choice of constants. He recommends taking a sample S of size $\log^{\frac{1}{3}} N \cdot (\frac{N}{B})^{\frac{2}{3}}$ from the input range `[first, last)`, and then partitioning `[first, last)` around values X_{low} and X_{high} which are taken from positions $[\frac{S}{N} \cdot n] - (\log N \cdot S)^{\frac{1}{2}}$ and $[\frac{S}{N} \cdot n] + (\log N \cdot S)^{\frac{1}{2}}$ in S . However, Sibeyn’s algorithm does not have special handling of cases where the n th element is at either extreme of `[first, last)`, i.e. at the first or last \sqrt{N} positions, for example. The `lazy_select` implementation handles such cases by testing whether the n th position is smaller than \sqrt{N} or larger than $N - \sqrt{N}$, and if so, partitions the input range into only two sections, one of which is passed to the deterministic `select` function that is used as subroutine throughout. Another deviation from Sibeyn’s basic design is that `lazy_select` is not recursive. Instead, only one

sample is drawn and based on that, the best of the deterministic implementations discussed earlier, namely `select_with_unrolled_median`, is called with the appropriate section of the partitioned range as input. The reason is my desire to guarantee an asymptotic complexity of $O(n)$, which is achieved with the current design since `lazy_select` is called only once with each invocation, and the subroutine (`select_with_unrolled_median`) has linear complexity, as described earlier.

Asymptotic complexity is one thing, runtime performance another. However, `lazy_select` measures up in this respect also. As appears from figure 6, `lazy_select` performs well with all types of iterator. Its performance with random access iterators (the `vector` structure) is of course better than with other iterators because random access iterators are inherently “cheaper” to manipulate than other iterators. Performance with forward and bidirectional iterators is still quite good, though, as can be ascertained by comparing with the `trivial_nth_element` implementation. The `lazy_select` function is almost as fast and requires only a fraction of the additional storage that `trivial_nth_element` allocates. Furthermore, when the input range contains large objects and not fundamental types such as `int`, `lazy_select` will most likely outperform `trivial_nth_element` since it avoids the latter’s costly copying operations.

One technical detail merits attention. In the specialization of `lazy_select` for forward and bidirectional iterators, the random sample is drawn while traversing the input range to determine the size of that range (N), and the index of the sought after element (n). The motivation for drawing the sample while at the same time determining the values of two needed parameters is to minimize the number of visits made. Obviously, it is more cost efficient to do both tasks in a single traversal than to make two separate passes over the input. It does present a problem in practice, though, since the size of the sample is dependent on the range size N which is only known after the traversal is complete! To circumvent this problem, I draw a sample whose size with a high degree of probability is $\frac{1}{32}N$ by assigning each iterator/element pair from the input range to the sample with probability $\frac{1}{32}$. This is carried out using only $\frac{1}{6}N$ random numbers by splitting a 32-bit random number generated every six iterations into six 5-bit sequences that are then used with the following six elements to decide which go into the sample. The choice of fractional constant, i.e. $\frac{1}{32}$, is dictated by the need to ensure that the size of the sample is equal to or greater than the size recommended by Sibeyn [9, p.295] without actually knowing the exact size beforehand.

This technique of combining passes over input to minimize the number of visits is also used in function `partition` when the input range is partitioned into the three sections described earlier. In that function, the distance from the beginning of the input range to the middle section as well as the size of the middle section itself are computed while partitioning the range. That way, separate calls to the `distance()` function can be avoided when the distance values (which are passed by reference) are returned to the calling `lazy_select` function. The combined effect of applying these performance engineering techniques proved quite substantial in improving runtime performance. For implementation details, see the `lazy_select` implementation and the `auxiliary functions` listing in the appendix.

lazy_select with different data structures

Compiled using g++ -O6. Executed on nanna.diku.dk.

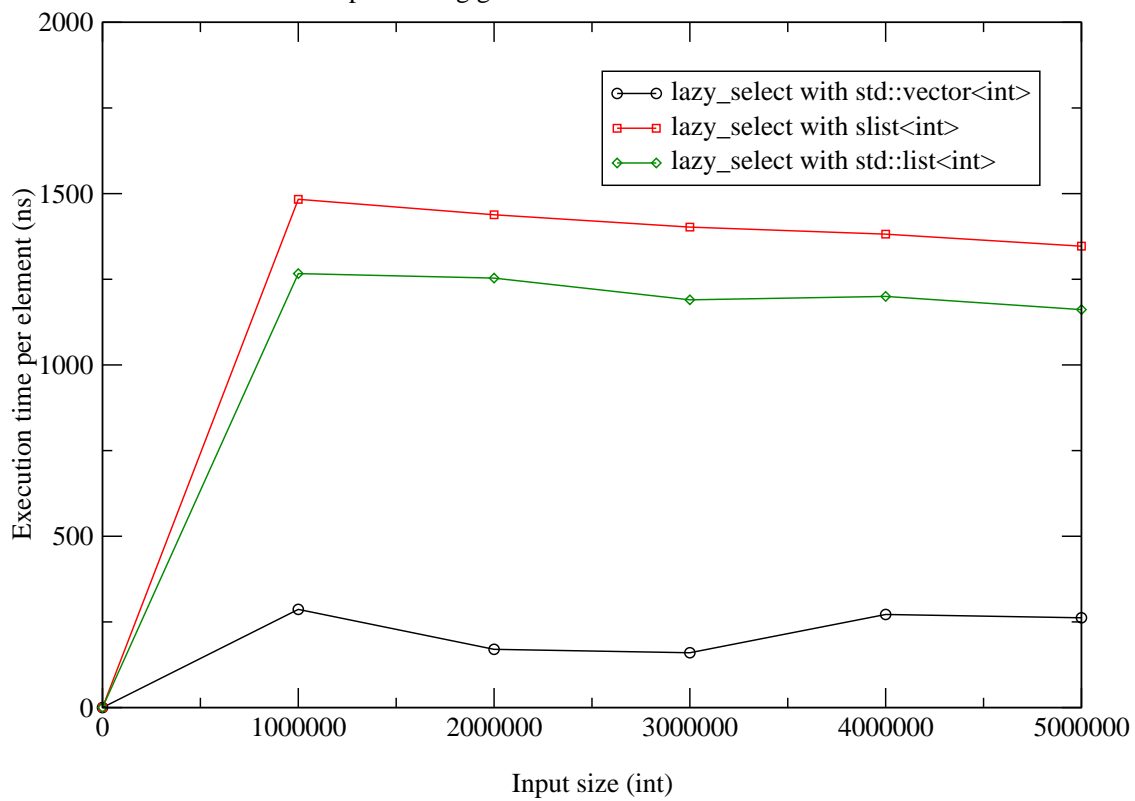


Figure 6: lazy_select execution times.

lazy_select with different data structures

Compiled using g++. Executed on nanna.diku.dk.

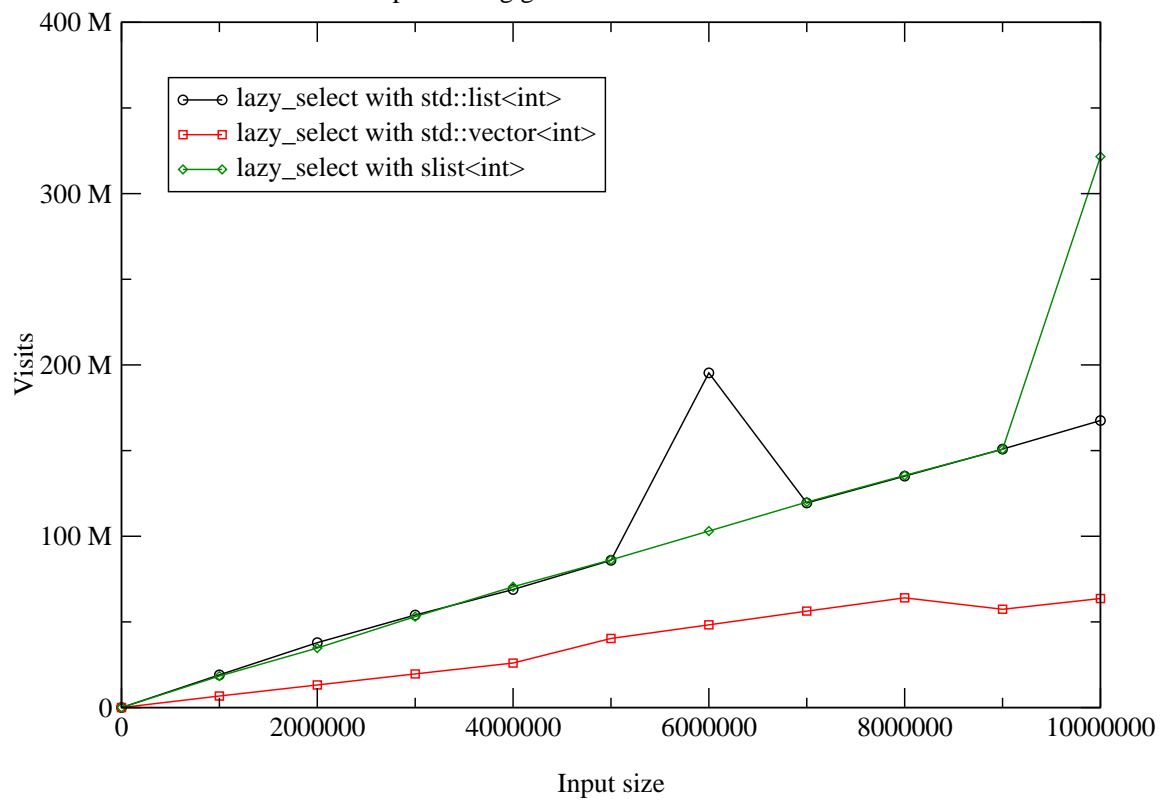


Figure 7: lazy_select visits.

9 Choosing an implementation

The ground has now been laid for deciding which implementation, if any, is to be recommended for inclusion into the Copenhagen STL.

Using a method of selection through elimination, the two worst-performing versions of `select`, namely `select_with_pq_distance_optimization` and `select_without_pq_distance_optimization`, can be quickly excluded from consideration. The reason, of course, is that they perform markedly worse in terms of execution time and visits than the most aggressively optimized version of `select`, i.e. `select_with_unrolled_median`.

The `randized_select` implementation has a theoretical worst-case complexity of $O(N^2)$, so on paper, it ought not to merit consideration at all, given the competition. However, in practice it performs quite well as appears from the benchmarking results in section 7. Furthermore, it is simple in design and implementation, operates in place and the underlying algorithm can be easily communicated to third party users.

`trivial_nth_element` is just that: trivial, both in design and implementation. Its runtime performance with data structures whose essential operations have linear time complexity is, however, interesting. When used with the STL `list` and `slist` structures, it outdoes other implementations through its simple tactic of simply circumventing the costly operations of those structures by moving the contents to cheaper storage (in a `vector`) for further processing. The drawback is the added $O(N)$ storage this requires – `trivial_nth_element` does not use an in-place algorithm, in other words. Also, since the implementation used in this case is the standard STL `nth_element` function, the worst-case complexity is $O(n \log n)$. This could be remedied by replacing `nth_element` with `select_with_unrolled_median`, but the performance figures would then not be as impressive, as can be deduced from the benchmark figures in section 6.

The `lazy_select` function has an asymptotic $O(N)$ complexity, giving it the desirable linear time guarantee we are after in our choice of `cphstl::nth_element` implementation. Its runtime performance is very good indeed, at least in the case of random access iterators. The performance with other types of iterator is not on a par but still comparable, and using `lazy_select` with `list` and `slist` structures is still markedly better than using `sort()` which is the only alternative if only `std::nth_element` is available.

In figures 8, 9, and 10 are shown the runtime performance of all the best performing implementations for easy comparison. The number of visits is commensurate to the execution times and can be deduced from figures elsewhere in this report.

10 Conclusion

All in all, the current implementation of `lazy_select` seems the best of the above candidates for inclusion into the Copenhagen STL as `cphstl::nth_element`. Overall, none has as many desirable traits as `lazy_select` (support for forward iterators and up, $O(N)$ complexity, fast for almost all uses), and the runtime performance is very good compared to the alternatives available.

Further work could lie in improving the adjustable parameters of `lazy_select`, e.g. the size of the proportional constant that governs how large the random sample is, and in further

Best performing implementations with `std::vector<int>`.

Compiled using `g++ -O6`. Executed on `nanna.diku.dk`.

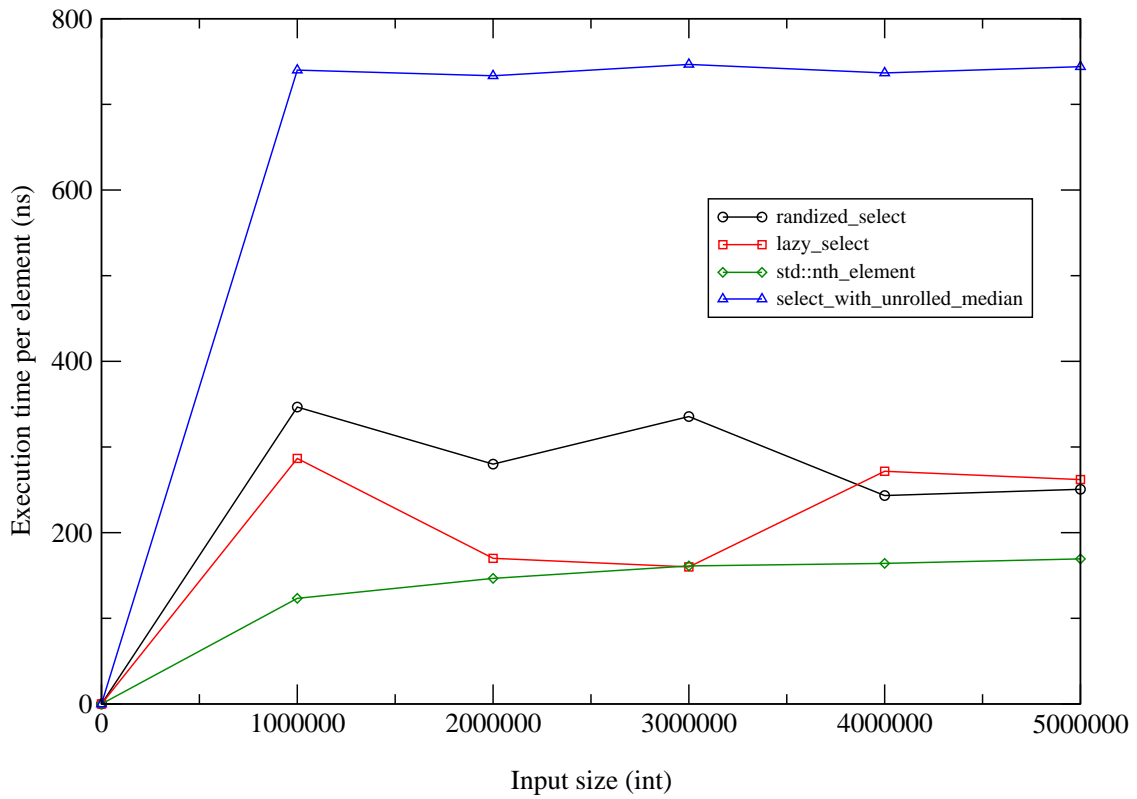


Figure 8: Best performing implementations with `vector<int>`

Best performing implementations with `std::list<int>`.

Compiled using `g++ -O6`. Executed on `nanna.diku.dk`.

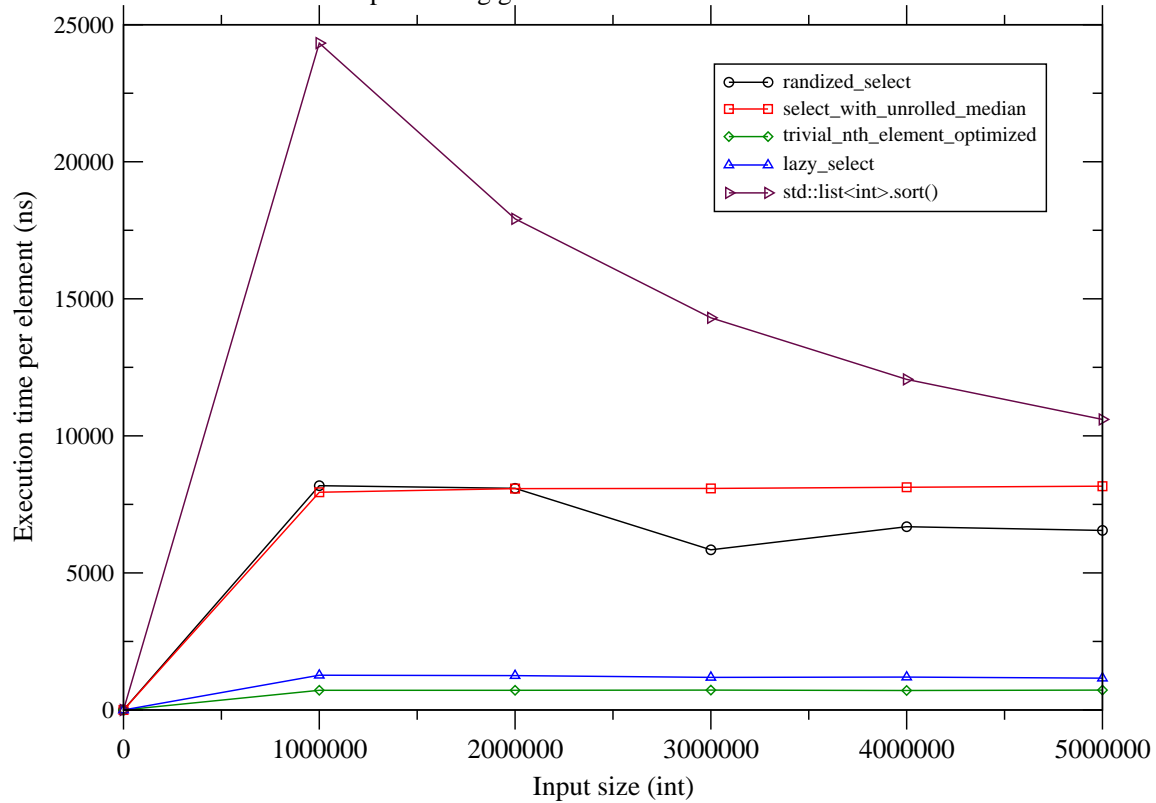


Figure 9: Best performing implementations with `list<int>`

Best performing implementations with `slist<int>`

Compiled using `g++ -O6`. Executed on `nanna.diku.dk`.

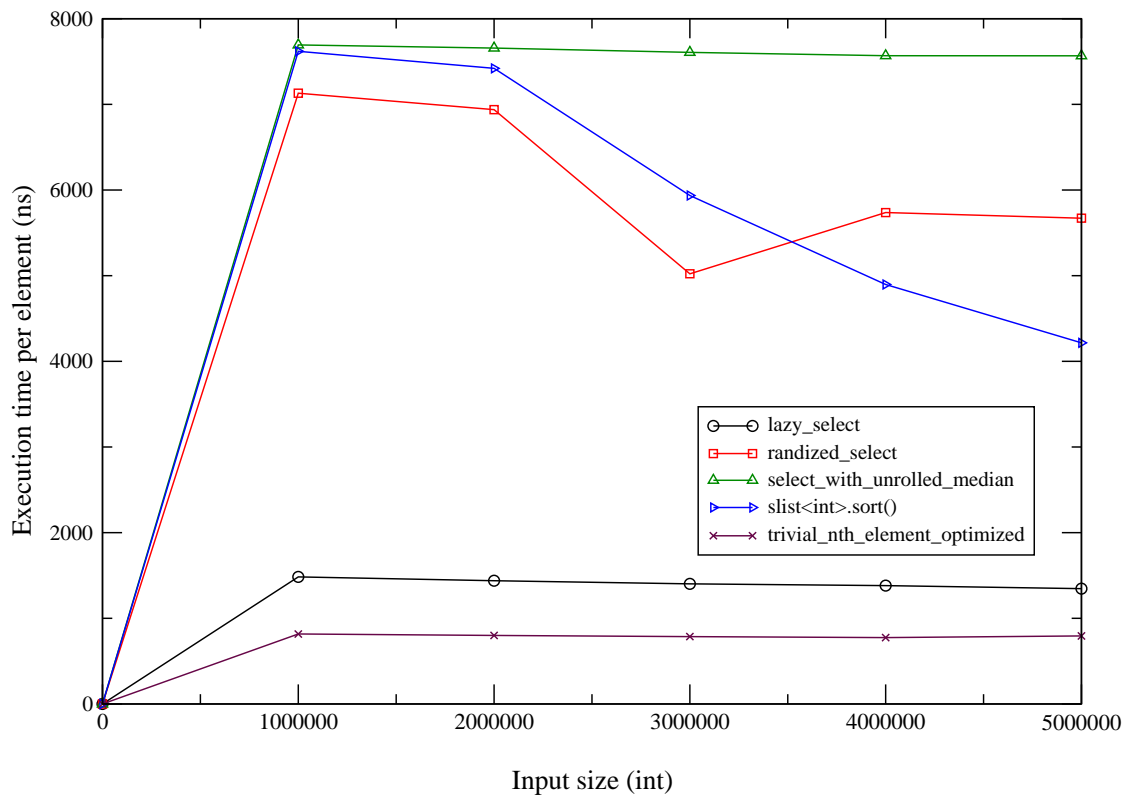


Figure 10: Best performing implementations with `slist<int>`

exploring Sibeyn's article[9]⁶ which has significantly influenced this work.

References

- [1] AMERICAN NATIONAL STANDARDS INSTITUTE, *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++: Document Number X3J16/96-0225 WG21/N1043* (1997). This is the first freely-available release of the Draft ANSI C++ Standard. This document is the second draft released for public review.
- [2] J. L. BENTLEY, *Programming Pearls*, second Edition, Addison-Wesley, Reading, MA, USA (1999).
- [3] M. BLUM, R. W. FLOYD, V. R. PRATT, R. L. RIVEST, AND R. E. TARJAN, Time bounds for selection, *Journal of Computer and System Sciences* **7**,4 (1973), 448–461.
- [4] J. BOJESEN, Managing memory hierarchies, M. Sc. Thesis, Department of Computing, University of Copenhagen (2000).
- [5] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, The MIT Press and McGraw-Hill Book Company (1989).
- [6] C. HOARE, I. HAYES, H. JIFENG, C. MORGAN, A. ROSCOE, J. SANDERS, I. SORENSEN, J. SPIVEY, B. SUFRIN, AND O PROGRAMMING, *Communications of the ACM* (1987).
- [7] J. KATAJAINEN AND T. PASANEN, A randomized in-place algorithm for positioning the kth element in a multiset, Worldwide Web Document (2001). Available at www.cphstl.dk.
- [8] R. MOTWANI AND P. RAGHAVAN, *Randomized Algorithms*, Cambridge University Press (1995).
- [9] J. F. SIBEYN, External selection, *Lecture Notes in Computer Science* **1563** (1999), 291–301.
- [10] PERFORMANCE ENGINEERING LABORATORY, The copenhagen stl website, Website accessible at <http://www.cphstl.dk> (2002).

Appendices

A Source code

Listings

1 "lazy_select implementation" 22

⁶By which I mean translating his design for a deterministic algorithm into an implementation, and looking into how the randomized algorithm holds up when it has to deal with particular types of input data, e.g. data having $n \leq \text{sqrt}(N)$.

2	"randized_select implementation"	27
3	"select implementation"	29
4	"trivial_nth_element implementation"	37
5	"Various helper functions"	38
6	"Benchmarking harness (main() code) for visits benchmarking "	43
7	"Benchmarking harness (main() code) for execution time benchmarking "	48
8	"Miscellaneous functions"	52
9	"Header for miscellaneous functions"	52
10	"Median-of-5 macro"	56

Listing 1: "lazy_select implementation"

```

/*! \ file
*/

#include <iterator>
#include <vector>
#include <functional>
#include <fstream>
#include <stdlib.h>
#include <time.h>

using namespace std;

const unsigned long INPUT_SIZE_THRESHOLD = 10000;
const unsigned long BLOCK_SIZE = 16;
const unsigned long SAMPLE_SIZE_THRESHOLD = 100;

/* \brief This functions draws a proportional sample of the range [ first , beyond).
   To minimize the runtime cost, the sampling only generates a random number once for every six
   elements. It then applies a bit mask to the 32-bit random number to make use of the six disjoint
   5-bit sequences contained in that number. Each 5-bit sequence is used to decide whether an
   element of the input range should be put into the sample. If the sequence is all 1's, the
   element is added as a pair consisting of the iterator pointing to the element in question, and
   the element itself . This practice is used to avoid further dereferencing of the iterator when
   the sample is used in the ongoing processing of the lazy_select function proper.
   Varying the choice of RNG doesn't affect runtime performance much (has experimented with
   ISAAC and Mersenne Twister),
   so plain old rand() is used.
*/
template < class position , class T >
void rnd_sample_fixed_proportion(    position first ,
                                     position nth,
                                     position beyond,
                                     std::vector< T >&
                                     fixed_proportion_sample,
                                     std::iterator_traits < position >::
                                     difference_type& first_nth_distance,
                                     std::iterator_traits < position >::
                                     difference_type&
                                     first_beyond_distance )
{
    // this call to srand() could be eliminated in any but the first call by using a static variable

```

```

        as flag
// or moving it outside this function. Such optimization is left as a future improvement
// since the entire random number generation can most likely be further improved by
// introducing a RNG customized for generating 5-bit sequences as are needed here.
srand( clock() );
typedef std:: iterator_traits< position >::value_type valtype;

unsigned int rand_mask = 0;
unsigned int rand_count = 0;
unsigned int rand_val = 0;
for ( first_beyond_distance = 0 ; first != beyond; first ++, first_beyond_distance++, visit(2)
)
{
    if ( rand_count == 0 )
    {
        rand_count = 6;
        rand_val = rand();
        rand_mask = 0x1f;
    }
    if ( (rand_val & rand_mask) == rand_mask )
    {
        fixed_proportion_sample.push_back( T( first, *first ) );
    }
    if ( first == nth )
    {
        first_nth_distance = first_beyond_distance;
    }
    rand_mask <<= 5;
    rand_count--;
}
}

```

```

template<class iterator , class element>
struct sample_pair
{
    sample_pair( iterator iter_arg , element& elem_arg ) : iter( iter_arg ) , elem( elem_arg ) {} ;
    inline bool operator<( const sample_pair& s1 )const
    {
        return elem < s1.elem;
    };
    iterator iter ;
    element elem;
};

```

/*! \ brief lazy_select implementation based on Randomized Algorithms, p.48, Sibeyn (External Selection),
 and Bojesens implementation in "Memory Hierarchies". This specialization for forward and bidirectional
 iterators .

```

*/
template< class iterator , class ordering >
void lazy_select( iterator first , iterator nth , iterator last , ordering less , std::
forward_iterator_tag )
{
    typedef std:: iterator_traits< iterator >::value_type element;
    typedef std:: iterator_traits< iterator >::difference_type difference ;
    difference N, n;
    const double block_size = BLOCK_SIZE / sizeof( element );

```

```

// sample proportionally across the input range to get raw material for further sampling
// and to compute essential parameters
vector< sample_pair<iterator, element> > fixed_sample;
rnd_sample_fixed_proportion( first, nth, last, fixed_sample, n, N );
// find the size of the sample to be derived using the parameter values just found
difference S = static_cast< difference >( ceil (pow (log ((double) N) * pow ((double) N /
block_size,
2.0),
1.0 / 3.0)));
// default to deterministic selection if below threshold
if ( N < INPUT_SIZE_THRESHOLD || S < SAMPLE_SIZE_THRESHOLD)
{
    benchmark_functions::select_with_unrolled_median( first, nth, last );
}
// if above threshold, proceed with lazy_select
else
{
    // compute essential parameters of the sibeyn algorithm
    difference delta = static_cast< difference >
( ceil (sqrt (log ((double) N) * S)) / 2);
    difference offset = static_cast< difference >( static_cast< double >( S ) * ( static_cast
< double >( n ) / static_cast< double >(N) ));
    // find sample_low and sample_high
vector< sample_pair<iterator,element> >::iterator s_low = delta < offset ? (fixed_sample.
begin() + (offset - delta)) : (fixed_sample.begin());
vector< sample_pair<iterator,element> >::iterator s_high = (offset + delta) > S ? (
fixed_sample.begin() + S - 1) : (fixed_sample.begin() + offset + delta);
lazy_select( fixed_sample.begin(), s_high, fixed_sample.begin() + S);
lazy_select( fixed_sample.begin(), s_low, s_high );

    iterator sample_low = s_low->iter;
    iterator sample_high = s_high->iter;

    visit (2);
    // if n below threshold, do 2-way partitioning rather than the standard 3-way fare
    difference margin = static_cast< difference >( pow( N, 0.5 ));
    if ( n < margin )
    {
        pair< iterator , iterator > middle_section = benchmark_functions::partition( first
, sample_high, last, less );
        if ( n < benchmark_functions::distance( first, middle_section.first ) )
        {
            benchmark_functions::select_with_unrolled_median( first, nth,
middle_section.first, less );
        }
        else
        {
            benchmark_functions::select_with_unrolled_median( middle_section.first
, nth, last, less );
        }
        return;
    }
    else if ( n > (N - margin) )
    {
        pair< iterator , iterator > middle_section = benchmark_functions::partition( first
, sample_low, last, less );
        if ( n < benchmark_functions::distance( first, middle_section.first ) )

```

```

        {
            benchmark_functions::select_with_unrolled_median( first, nth,
                middle_section.first, less );
        }
        else
        {
            benchmark_functions::select_with_unrolled_median( middle_section.first
                , nth, last, less );
        }
        return;
    }
    difference first_to_middle_start, middle_section_size;
    pair< iterator , iterator > middle_section = benchmark_functions::partition( first, last , *
        sample_low, *sample_high, first_to_middle_start, middle_section_size, less );
    // process the different cases that may be triggered by different configurations
    // of the above parameters
    difference dist;
    if ( n < first_to_middle_start )
    {
        benchmark_functions::select_with_unrolled_median( first, nth,
            middle_section.first, less );
    }
    else if ( n > first_to_middle_start + middle_section_size )
    {
        benchmark_functions::select_with_unrolled_median( middle_section.
            second, nth, last, less );
    }
    else
    {
        benchmark_functions::select_with_unrolled_median( middle_section.first
            , nth, middle_section.second, less );
    }
}
}

```

/* \brief lazy_select implementation based on Randomized Algorithms, p.48, Sibeyn (External Selection), and Bojesens implementation in "Memory Hierarchies". This specialization for random access iterators.

*/

```

template < class iterator , class ordering >
void lazy_select( iterator first , iterator nth, iterator last , ordering less , std::
    random_access_iterator_tag )
{

```

```

    typedef std::iterator_traits< iterator >::value_type element;
    typedef std::iterator_traits< iterator >::difference_type difference;

```

```

    difference N, n;
    N = last - first;
    n = nth - first;

```

```

    const double block_size = BLOCK_SIZE / sizeof( element );
    difference S = static_cast < difference > ( ceil ( pow ( log ((double) N) *
        pow ((double) N /
            block_size,

```

```

                2.0),
                1.0 / 3.0));
// default to deterministic selection if below threshold
if ( N < INPUT_SIZE_THRESHOLD || S < SAMPLE_SIZE_THRESHOLD )
{
    benchmark_functions::select_with_unrolled_median( first, nth, last );
}
// if above threshold, proceed with lazy_select
else
{
    // compute essential parameters of the sibeyn algorithm

    difference delta = static_cast< difference >
( ceil (sqrt (log ((double) N) * S)) / 2);

    difference offset = static_cast< difference >( static_cast< double >( S ) * ( static_cast<
double >( n ) / static_cast< double >(N) ));

    // find sample_low and sample_high
    iterator sample_low = delta < offset ? first + (offset - delta) : first;
    iterator sample_high = (offset + delta) > S ? first + S - 1 : first + offset + delta;
    visit (2);
// if n below threshold, do 2-way partitioning rather than the standard 3-way fare
    difference margin = static_cast< difference >( pow( N, 0.5 ));
    if ( n < margin )
    {
        pair< iterator, iterator > middle_section = benchmark_functions::partition( first
, sample_high, last, less );
        if ( nth < middle_section.first )
        {
            benchmark_functions::select_with_unrolled_median( first, nth,
middle_section.first, less );
        }
        else
        {
            benchmark_functions::select_with_unrolled_median( middle_section.first
, nth, last, less );
        }
        return;
    }
    else if ( n > (N - margin) )
    {
        pair< iterator, iterator > middle_section = benchmark_functions::partition( first
, sample_low, last, less );
        if ( nth < middle_section.first )
        {
            benchmark_functions::select_with_unrolled_median( first, nth,
middle_section.first, less );
        }
        else
        {
            benchmark_functions::select_with_unrolled_median( middle_section.first
, nth, last, less );
        }
        return;
    }
}
// else, draw a sample and partition around sample_low, sample_high

```

```

benchmark_functions::rnd_sample( first, last, S, N );

benchmark_functions::select_with_unrolled_median( first, sample_high, first + S, less );
benchmark_functions::select_with_unrolled_median( first, sample_low, sample_high, less );
pair< iterator , iterator > middle_section = benchmark_functions::partition( first, last , *
    sample_low, *sample_high, less );

#ifdef DEBUG
output « "vector middle first " « benchmark_functions::distance( first , middle_section.
    first ) « endl;
output « "vector middle second" « benchmark_functions::distance( first , middle_section.
    second ) « endl;
#endif
// process the different cases that may be triggered by different configurations
// of the above parameters
if ( n < benchmark_functions::distance( first, middle_section.first ) )
{
    benchmark_functions::select_with_unrolled_median( first, nth,
        middle_section.first, less );
}
else if ( n > benchmark_functions::distance( first, middle_section.second ) )
{
    benchmark_functions::select_with_unrolled_median( middle_section.
        second, nth, last, less );
}
else
{
    benchmark_functions::select_with_unrolled_median( middle_section.first
        , nth, middle_section.second, less );
}
}

}

/*! \brief lazy_select dispatch
*/
template < class iterator , class ordering >
void lazy_select( iterator first , iterator nth, iterator last , ordering less )
{
    lazy_select( first , nth, last , less , std::iterator_traits< iterator >::iterator_category() );
}

```

Listing 2: "randized_select implementation"

```

template < class item > item pivot (item p, item r)
{
    std::iterator_traits < item >::difference_type randnum =
        static_cast < std::iterator_traits < item >::difference_type >
        ((benchmark_functions::
            distance (p,
                r) * (static_cast < double >(rand ()) / static_cast <
                    double >(RAND_MAX))));
    std::advance (p, randnum);
    return p;
}

// partition for forward iterators and up
/* We want a range partitioned like this:

```

```

p1p2p3 q1q2q3 r1r2r3 ?
where less(p?, q?) && (!less(q?,q?) && !less(q?',q?)) && !less(r?, q?)
using ++ as the only form of incrementation.
? represents the element under consideration.
If ? belongs to the X section, rotate( z1, y1, ? )
If ? belongs to the Y section, swap( z1, ? )
If ? belongs to the Z section, continue
*/
template < class item, class ordering >
item partition_rselect (item p, item q, item r, ordering less,
                      std::forward_iterator_tag)
{
    item Q_first = p;
    item R_first = p;
    std::iterator_traits < item >::value_type value = *q;

    while (p != r)
    {
        // P or Q-section
        if (!less (*p, value))
        {
            // Q-section
            if (!less (value, *p))
            {
                swap (*p, *R_first);
                R_first++; visit(1);
            }
        }
        else // R-section
        {
            benchmark_functions::rotate3 (p, R_first++, Q_first++); visit(2);
        }
        p++; visit(1);
    }
    return Q_first;
}

template < class item, class ordering >
item partition_rselect (item p, item q, item r, ordering less,
                      std::random_access_iterator_tag)
{
    iterator_traits < item >::value_type x = *q;
    swap (*p, *q);
    item i = p;
    item j = r;
    for (;;)
    {
        do {
            i++;
            visit (1);
        }
        while (i < r && less (*i, x));
        do {
            j--;
            visit (1);
        }
        while (less (x, *j));
    }
}

```

```

        if ( i < j )
        {
            swap (*i, *j);
        }
        else
        {
            swap (*p, *j);
            return j;
        }
    }
}

// partition dispatch
template < class item, class ordering >
    inline item partition_rselect (item p, item q, item r, ordering less)
    {
        return partition_rselect (p, q, r, less,
                                   std::iterator_traits <
                                   item >::iterator_category ());
    }

template < class item, class ordering >
    void randized_select (item p, item q, item r, ordering less)
    {
        std::iterator_traits < item >::difference_type diff =
            benchmark_functions::distance (p, r);
        if ( diff > 1 )
        {
            item p_prime = pivot (p, r);
            item q_prime = partition_rselect (p, p_prime, r, less);
            if ( benchmark_functions::distance (p, q) <
                  benchmark_functions::distance (p, q_prime) )
                randized_select (p, q, q_prime, less);
            else if ( benchmark_functions::distance (p, q) >
                       benchmark_functions::distance (p, q_prime) )
            {
                randized_select (++q_prime, q, r, less); visit (1);
            }
        }
    }

template < class item > inline void randized_select (item p, item q, item r)
    {
        typedef iterator_traits < item >::value_type T;
        return randized_select (p, q, r, less < T > ());
    }

```

Listing 3: "select implementation"

```

/*! \ file
* \ brief implementation of the Blum et al. PICK algorithm
*
* <h3>Original author</h3>
* Jyrki Katajainen
*
* <h3>Later modifications</h3>
* Lars Yde

```

```

*/

// original insertion sort by Jyrki Katajainen. Modified for use with
// forward iterators and up by Lars Yde.
// The circuitous code style is due to the fact that only ++ can be used
// to advance iterators.
template < class item, class ordering >
void insertion_sort (item p, item r, ordering less)
{
    std::iterator_traits < item >::difference_type count, start, length, diff;
    item i, k;
    item j = p;
    diff = benchmark_functions::distance (j, r);
    for (count = 1; count < diff; count++)
    {
        j++;
        visit (1);
        iterator_traits < item >::value_type key = *j;
        length = benchmark_functions::distance (p, j);
        // Note: the below may seem more clunky than Jyrki's original code
        // but I arranged it this way to avoid passing negative values to
        // advance as this causes a segmentation fault with forward iterators.
        for (start = 1; start <= length; start++)
        {
            i = p;
            std::advance (i, length - start);
            if (less (key, *i))
            {
                k = i;
                k++;
                visit (1);
                *k = *i;
            }
            else
            {
                i++;
                visit (1);
                break;
            }
        }

        *i = key;
    }
}

// insertion sort convenience function
template < class item > inline void insertion_sort (item p, item r)
{
    typedef iterator_traits < item >::value_type T;
    benchmark_functions::insertion_sort (p, r, less < T > ());
}

// partition for forward iterators and up
/* We want a range partitioned like this:
p1p2p3 q1q2q3 r1r2r3 ?
where less(p?, q?) && (!less(q?,q?) && !less(q?,q?)) && !less(r?, q?)

```

```

    using ++ as the only form of incrementation.
    ? represents the element under consideration.
    If ? belongs to the X section, rotate( z1, y1, ? )
    If ? belongs to the Y section, swap( z1, ? )
    If ? belongs to the Z section, continue
*/
template < class item, class ordering >
    pair < item, item > partition (item p, item q, item r, ordering less,
                                std::forward_iterator_tag)
{
    item Q_first = p;
    item R_first = p;
    std::iterator_traits < item >::value_type value = *q;

    while (p != r)
    {
        // P or Q-section
        if (!less (*p, value))
        {
            // Q-section
            if (!less (value, *p))
            {
                swap (*p, *R_first);
                R_first++;
                visit (1);
            }
        }
        else // R-section
        {
            rotate3 (p, R_first++, Q_first++); visit(2);
        }
        p++; visit(1);
    }
    // return a pair of iterators designating the range [q1..qn+1)
    return pair < item, item > (Q_first, R_first);
}

// partition for random access iterators (adapted from version by Katajainen)
template < class item, class ordering >
    pair < item, item > partition (item p, item q, item r,
                                ordering less,
                                std::random_access_iterator_tag)
{
    typedef std::iterator_traits < item >::difference_type Difftype;
    iterator_traits < item >::value_type v = *q;
    swap(*p, *q);
    item pa = p + 1;
    item pb = p + 1;
    item pc = r - 1;
    item pd = r - 1;
    for (;;) {
        while (pb <= pc && !less(v, *pb)) {
            if (less(*pb, v))
            {
                pb += 1; visit(1);
            }
            else {

```

```

        swap(*pa, *pb);
        pa += 1; visit(1);
        pb += 1; visit(1);
    }
}
while (pb <= pc && !less(*pc, v)) {
    if (less(v, *pc))
    {
        pc --;
        visit(1);
    }
    else {
        swap(*pc, *pd);
        pd --; visit(1);
        pc --; visit(1);
    }
}
if (pb > pc)
    break;
swap(*pb, *pc);
pb += 1; visit(1);
pc --; visit(1);
}
Difftype smaller = pb - pa;
Difftype i = min(Difftype(pa - p), smaller);
item k;
item l;
for (k = p, l = pb - i; l < pb; ++k, ++l, visit(2) )
    swap(*k, *l);
Difftype larger = pd - pc;
Difftype j = min(larger, Difftype(r - pd - 1));
for (k = pb, l = r - j; l < r; ++k, ++l, visit(2) )
    swap(*k, *l);
return pair<item, item>(p + smaller, r - larger);

}

// partition convenience function
template < class item >
    inline pair < item, item > partition (item p, item q, item r,
                                        std::bidirectional_iterator_tag)
{
    typedef iterator_traits < item >::value_type T;
    return partition (p, q, r, less < T > ());
}

// partition dispatch
template < class item >
    inline pair < item, item > partition (item p, item q, item r)
{
    return partition (p, q, r,
                    std::iterator_traits < item >::iterator_category ());
}

// partition dispatch
template < class item, class ordering >
    inline pair < item, item > partition (item p, item q, item r,

```

```

                                ordering less)
{
    return partition (p, q, r, less,
                    std::iterator_traits < item >::iterator_category ());
}

template < class item, class ordering,
          std::iterator_traits < item >::difference_type gs > void select (item p,
                                item q,
                                item r,
                                std::iterator_traits < item >::
                                    difference_type p_q_distance,
                                ordering
                                less)
{
    if (benchmark_functions::distance (p, r) < 90)
    {
        insertion_sort (p, r, less);
    }
    else
    {
        item i = p;
        item j;
        std::iterator_traits < item >::difference_type diff =
            benchmark_functions::distance (p, r);
        std::iterator_traits < item >::difference_type start;

        for (j = p, start = 0; start < diff - gs;
             ++i, visit (1), advance (j, gs), start += gs)
        {
            item block_end = j;
            advance (block_end, gs);
            visit (1);
            insertion_sort (j, block_end, less);
            item block_median = j;
            advance (block_median, gs / 2);
            visit (1);
            swap (*i, *(block_median));
        }
        item p_prime = p;
        std::advance (p_prime, benchmark_functions::distance (p, i) / 2);
        select < item, ordering, gs > (p, p_prime, i, less);
        pair < item, item > z = partition (p, p_prime, r, less);

        if ( p_q_distance <=
            benchmark_functions::distance (p, z.first) )
        {
            select < item, ordering, gs > (p, q, z.first, p_q_distance, less);
        }
        else if ( p_q_distance >=
                 benchmark_functions::distance (p, z.second) )
        {
            select < item, ordering, gs > (z.second, q, r, benchmark_functions::distance (z.second, q),
                                           less);
        }
    }
}

```

```

}

/* call an overloaded version of select that takes the distance between
p and q as an extra parameter so as to avoid recomputing that value
with each recursive call. This trick only applies to those recursive
calls where the value of p does not change (the value of q never changes), but that means half on
average and so makes for a worthwhile savings at little expense.
*/
template < class item, class ordering,
std::iterator_traits < item >::difference_type gs > void select (item p,
                                                                item q,
                                                                item r,
                                                                ordering
                                                                less)
{
    select < item, ordering, gs > ( p, q, r, benchmark_functions::distance( p, q ), less );
}

// select convenience function
template < class item, size_t gs >
inline void select (item p, item q, item r)
{
    typedef iterator_traits < item >::value_type T;
    select < item, less < T >, gs > (p, q, r, less < T > ());
}

template < class item, class ordering,
std::iterator_traits < item >::difference_type gs > void select_without_pq_distance_optimization (
    item p,
                                                                item
                                                                q
                                                                ,
                                                                item
                                                                r
                                                                ,
                                                                ordering
                                                                less )
{
    if (benchmark_functions::distance (p, r) < 90)
    {
        insertion_sort (p, r, less);
    }
    else
    {
        item i = p;
        item j;
        std::iterator_traits < item >::difference_type diff =

```

```

    benchmark_functions::distance (p, r);
std::iterator_traits < item >::difference_type start;

for (j = p, start = 0; start < diff - gs;
    ++i, advance (j, gs), start += gs)
{
    item block_end = j;
    advance (block_end, gs);
    insertion_sort (j, block_end, less);
    item block_median = j;
    advance (block_median, gs / 2);
    swap (*i, *(block_median));
}
item p_prime = p;
std::advance (p_prime, benchmark_functions::distance (p, i) / 2);
select_without_pq_distance_optimization < item, ordering, gs > (p, p_prime, i, less);
pair < item, item > z = partition (p, p_prime, r, less);
// equivalent to distance( q, z.first ) >= 0, but compatible with
// the bidirectional "wrap-around" iterators in the SGI list<T> structure
if (benchmark_functions::distance (p, q) <=
    benchmark_functions::distance (p, z.first))
{
    select_without_pq_distance_optimization < item, ordering, gs > (p, q, z.first, less);
}
else if (benchmark_functions::distance (p, q) >=
    benchmark_functions::distance (p, z.second))
{
    select_without_pq_distance_optimization < item, ordering, gs > (z.second, q, r, less);
}
}

// select_without_pq_distance_optimization convenience function
template < class item, size_t gs >
inline void select_without_pq_distance_optimization (item p, item q, item r)
{
    typedef iterator_traits < item >::value_type T;
    select_without_pq_distance_optimization < item, less < T >, gs > (p, q, r, less < T > ());
}

// select with block size 5 and specialized sort
template < class item, class ordering > void select_with_unrolled_median (item p,
                                                                    item q,
                                                                    item r,
                                                                    std::iterator_traits < item >::
                                                                    difference_type p_q_distance,
                                                                    ordering
                                                                    less)
{
    if (benchmark_functions::distance (p, r) < 90)
    {
        insertion_sort (p, r, less);
    }
    else
    {

```

```

typedef std::iterator_traits< item >::value_type Valtype;
item i = p;
item j;
std::iterator_traits< item >::difference_type diff =
    benchmark_functions::distance (p, r);
std::iterator_traits< item >::difference_type start;

for (j = p, start = 0; start < diff - 5;
     ++i, start += 5)
{
    item block_median;
    item j1 = j++;
    item j2 = j++;
    item j3 = block_median = j++;
    item j4 = j++;
    item j5 = j++;
    Valtype v1 = *j1, v2 = *j2, v3 = *j3, v4 = *j4, v5 = *j5;
    median5(v1, v2, v3, v4, v5, block_median, less);
    swap (*i, *(block_median));
}

item p_prime = p;
std::advance (p_prime, benchmark_functions::distance (p, i) / 2);
select_with_unrolled_median< item, ordering > (p, p_prime, i, less);
pair< item, item > z = partition (p, p_prime, r, less);

if ( p_q_distance <=
    benchmark_functions::distance (p, z.first ))
{
    select_with_unrolled_median< item, ordering > (p, q, z.first, p_q_distance, less);
}
else if ( p_q_distance >=
    benchmark_functions::distance (p, z.second))
{
    select_with_unrolled_median< item, ordering > (z.second, q, r, benchmark_functions::
        distance( z.second, q ), less);
}
}
}

/* call an overloaded version of select that takes the distance between
p and q as an extra parameter so as to avoid recomputing that value
with each recursive call. This trick only applies to those recursive
calls where the value of p does not change (the value of q never changes), but that means half on
average and so makes for a worthwhile savings at little expense.
*/
template< class item, class ordering > void select_with_unrolled_median (item p,
                                                                    item q,
                                                                    item r,
                                                                    ordering
                                                                    less)
{
    select_with_unrolled_median< item, ordering > ( p, q, r, benchmark_functions::distance( p, q ), less )
    ;
}

```

```

// select_with_unrolled_median convenience function
template < class item >
    inline void select_with_unrolled_median (item p, item q, item r)
    {
        typedef iterator_traits < item >::value_type T;
        select_with_unrolled_median < item, less < T > > (p, q, r, less < T > ());
    }

```

Listing 4: "trivial_nth_element implementation"

```

/*! \ file
 * \ brief This file contains the definition of a trivial_nth_element function specialization for forward
 * \ iterators .
 * <h3>Original author</h3>
 * Lars Yde, april 2001
 *
 */

#include <algorithm>
#include <iterator>
#include <vector>
#include <functional>

template< class Iterator >
void trivial_nth_element_optimized( Iterator first, Iterator nth, Iterator last )
{
    typedef std::iterator_traits< Iterator >::value_type Valtype;
    typedef std::iterator_traits< Iterator >::difference_type Difftype;
    std::vector< Valtype > v;

    Difftype diff, count;
    Iterator iter;
    for ( iter = first, count = 0; iter != last; iter++, count++, visit(2) )
    {
        v.push_back( *iter );
        if ( iter == nth )
        {
            diff = count;
            // break from this loop and continue in the next to avoid superfluous if's and count++'s
            break;
        }
    }
    // continue from above loop if iter == nth became true before end of sequence was reached
    for ( iter++, visit(1); iter != last; iter++, visit(1) )
    {
        v.push_back( *iter );
    }
    std::vector< Valtype >::iterator v_nth = v.begin();
    std::advance( v_nth, diff );
    // find nth element in random access sequence
    lazy_select( v.begin(), v_nth, v.end(), std::less < Valtype >() );
    // copy back to forward access iterator sequence
    std::copy( v.begin(), v.end(), first );
}

template< class Iterator >

```

```

void trivial_nth_element_plain( Iterator first , Iterator nth, Iterator last )
{
    typedef std::iterator_traits< Iterator >::value_type Valtype;
    typedef std::iterator_traits< Iterator >::difference_type Difftype;
    std::vector< Valtype > v;

    Difftype diff , count;
    Iterator iter ;
    for ( iter = first , count = 0; iter != last ; iter ++, count ++, visit(2) )
    {
        v.push_back( *iter );
        if ( iter == nth )
        {
            diff = count;
        }
    }
    std::vector< Valtype >::iterator v_nth = v.begin();
    std::advance( v_nth, diff );
    // find nth element in random access sequence
    lazy_select( v.begin(), v_nth, v.end(), std::less< Valtype >() );
    // copy back to forward access iterator sequence
    std::copy( v.begin(), v.end(), first );
}

```

Listing 5: "Various helper functions"

```

/*! \ file
    \brief various functions that serve auxiliary purposes in the nth_element implementations
*/

//3-way partition for forward and bidirectional iterators by Lars Yde
/* We want a range partitioned like this:
    X1X2X3 Y1Y2Y3 Z1Z2Z3 ?
    where less(X?, pivot1) && ( !less(Y?, pivot1) && !less(pivot2, Y?) ) && !less(Z?,pivot2)
    so we iterate from first to beyond and do the following:
    if ? in X, rotate(Z1,Y1,?)
    if ? in Y, swap(Z1,?)
    if ? in Z, continue
*/
template < class position , class element, class ordering >
pair < position , position > inline partition ( position first ,
                                             position beyond,
                                             element pivot1,
                                             element pivot2,
                                             std::iterator_traits< position >::difference_type&
                                             first_to_middle_start,
                                             std::iterator_traits< position >::difference_type&
                                             middle_start_to_end,
                                             ordering less ,
                                             std::forward_iterator_tag)
{
    first_to_middle_start = 0;
    middle_start_to_end = 0;
    position Y_first = first ;
    position Z_first = first ;
    while ( first != beyond)
    {

```

```

    if (! less (* first , pivot1))
    {
        if (! less (pivot2, * first ))
        {
            middle_start_to_end++;
            swap(*Z_first, * first );
            Z_first++; visit(1);
        }
    }
    else
    {
        first_to_middle_start++;
        rotate3( first , Z_first, Y_first);
        Z_first++; visit(1);
        Y_first++; visit(1);
    }
    first ++; visit(1);
}
return pair < position, position > (Y_first, Z_first);
}

//Bojesen/Katajainen 3-way partition for random access iterators
template < class position, class element, class ordering >
    pair < position, position > inline partition (position first ,
                                                position beyond,
                                                element pivot1,
                                                element pivot2,
                                                ordering less ,
                                                std::
                                                random_access_iterator_tag)
{
    position less_p1 = first ;
    position less_p2 = first ;
    position greater_p2 = beyond - 1;
    position greater_p1 = beyond - 1;
    while (true)
    {
        while ( less (*less_p1, pivot1))
            {less_p1++; visit(1); }
        while ( less (pivot2, *greater_p2))
            {greater_p2--; visit(1); }
        if (less_p1 >= greater_p2)
            break;
        if ( less (pivot2, *less_p1))
            if ( less (*greater_p2, pivot1))
            {
                iter_swap (less_p1++, greater_p2--);
                visit (2);
            }
        else
        {
            rotate3 (greater_p2--, less_p1++, less_p2++);
            visit (3);
        }
        else if ( less (*greater_p2, pivot1))
        {
            rotate3 (less_p1++, greater_p2--, greater_p1--);

```

```

        visit (3);
    }
    else
    {
        iter_swap (greater_p1--, greater_p2--); visit(2);
        iter_swap (less_p1++, less_p2++); visit(2);
    }
};
if (less_p1 == greater_p2)
{
    less_p1++; visit(1);
    greater_p2--; visit(1);
};

swap_ranges (first,
             min (less_p2, greater_p2 + 1 - (less_p2 - first)),
             max (less_p2, greater_p2 + 1 - (less_p2 - first)));
swap_ranges (less_p1,
             min (greater_p1 + 1, beyond - (greater_p1 + 1 - less_p1)),
             max (greater_p1 + 1, beyond - (greater_p1 + 1 - less_p1)));

return pair < position, position > (greater_p2 + 1 - (less_p2 - first),
                                   beyond - (greater_p1 + 1 - less_p1));
}

// dispatch function for partitioning
template < class position, class element, class ordering >
pair < position, position > inline partition (position first,
                                             position beyond,
                                             element pivot1,
                                             element pivot2,
                                             ordering less)
{
    return partition ( first, beyond, pivot1, pivot2, less,
                     std::iterator_traits <
                     position >::iterator_category ());
}

// dispatch function for partitioning with distance computation
template < class position, class element, class ordering >
pair < position, position > inline partition (position first,
                                             position beyond,
                                             element pivot1,
                                             element pivot2,
                                             std::iterator_traits < position >::difference_type&
                                             first_to_middle_start,
                                             std::iterator_traits < position >::difference_type&
                                             middle_start_to_end,
                                             ordering less)
{
    return partition ( first, beyond, pivot1, pivot2, first_to_middle_start, middle_start_to_end, less,
                     std::iterator_traits <
                     position >::iterator_category ());
}

// random_sample for random access iterators by Bojesen/Katajainen

```

```

template < class position >
void random_sample (position first, position beyond,
                   long samplesz, std:: iterator_traits < position >::difference_type diff, std::
                   random_access_iterator_tag tag)
{
    const position sample_end = first + samplesz;
    for ( ; first < sample_end; first++, visit(1) )
    {
        iter_swap ( first,
                   first +
                   ((unsigned
                    int) (( beyond - first ) * ((double) rand () / RAND_MAX))));
    }
}

```

```

// generates a sample of expected length samplesz
template < class position >
void random_sample (position first, position beyond,
                   long samplesz, std:: iterator_traits < position >::difference_type diff, std::
                   forward_iterator_tag tag)
{
    std:: iterator_traits < position >::difference_type start;
    position sample_iterator = first;
    for ( start = 0; start < diff; start++, first++, visit(2) )
    {
        // if random number less than sample length, swap into sample
        if (static_cast <
            long
            >(
                (diff *
                 (static_cast < double >(rand ()) / static_cast <
                  double >(RAND_MAX)))) < samplesz)
        {
            iter_swap (sample_iterator++, first); visit (1);
        }
    }
}

```

```

// random_sample dispatch
template < class position >
void rnd_sample (position first, position beyond,
                std:: iterator_traits <
                position >::difference_type samplesz,
                std:: iterator_traits <
                position >::difference_type diff)
{
    random_sample (first, beyond, samplesz, diff,
                  std:: iterator_traits < position >::iterator_category ());
}

```

```

template < class item, class ordering >
bool partitioned (item p, item q, item r, ordering less)
{
    for (item i = p; i != q; ++i, visit (1) )

```

```

    {
        if ( less (*q, *i) )
            {
                return false;
            }
    }
    item i = q;
    for ( i++, visit(1); i != r; ++i, visit(1) )
        {
            if ( less (*i, *q) )
                {
                    return false;
                }
        }
    return true;
}

```

```

template < class item > inline bool partitioned ( item p, item q, item r )
{
    typedef iterator_traits < item >::value_type T;
    return partitioned ( p, q, r, less < T > ());
}

```

```

// rotate arguments a,b,c one position to the left
template < class position >
void inline rotate3 ( position a, position b, position c )
{
    iterator_traits < position >::value_type tmp = *a;
    *a = *b;
    *b = *c;
    *c = tmp;
}

```

```

template < class item, class ordering >
bool sorted ( item p, item r, ordering less )
{
    item q = p;
    std::iterator_traits < item >::difference_type diff =
        benchmark_functions::distance ( p, r );
    for ( long i = 1; i < diff; i++ )
        {
            item q_ = q;
            q++;
            if ( less (*q, *(q_)) )
                return false;
        }
    return true;
}

```

```

template < class item > inline bool sorted ( item p, item r )
{
    typedef iterator_traits < item >::value_type T;
    return sorted ( p, r, less < T > ());
}

```

/* cphstl version of the distance iterator primitive. By Lars Yde and Jyrki Katajainen.

```

*/
/* The forward iterator version could be implemented by incrementing both the p and q pointers till one
   reaches the other, but unfortunately, this doesn't work with the SGI slist structure since it does
   not terminate with a loop back to the end node but rather some singleton structure. Applying
   operator ++ to this structure causes a runtime error so the forward iterator version is
   implemented more naively, namely by presuming that p precedes q and then incrementing p till q is
   reached. This is clearly not satisfactory and a more appropriate solution should be found.
*/
template < class position >
    inline std::iterator_traits <
        position >::difference_type distance ( position p, position q,
                                                std::forward_iterator_tag)
{
    std::iterator_traits < position >::difference_type return_value;
    for (return_value = 0; p != q; return_value++, p++, visit(2));
    return return_value;
}

// specialization of distance for bidirectional iterators
// it is assumed that p precedes q - a necessary prerequisite since this
// specialization must work with the SGI list structure which uses
// wrap-around iterators.
template < class position >
    inline std::iterator_traits <
        position >::difference_type distance ( position p, position q,
                                                std::bidirectional_iterator_tag)
{
    position p_copy = p;
    position q_copy = q;
    std::iterator_traits < position >::difference_type return_value = 0;

    for (return_value = 0; q != p_copy && p != q_copy;
         return_value++, p++, q--, visit(2) )
        ;
    return return_value;
}

// specialization of distance for random access iterators
template < class position >
    inline std::iterator_traits <
        position >::difference_type distance ( position p, position q,
                                                std::random_access_iterator_tag)
{
    return std::distance (p, q);
}

// distance dispatch function
template < class position >
    inline std::iterator_traits <
        position >::difference_type distance ( position p, position q)
{
    return benchmark_functions::distance (p, q,
                                          std::iterator_traits <
                                              position >::iterator_category ());
}

```

Listing 6: "Benchmarking harness (main() code) for visits benchmarking "

```

/* Description: Harness for running benchmark tests on nth_element implementations.

Author: Lars Yde (larsyde@diku.dk)

Revision history: 28/11/2000 – coding begun
                  19/12/2000 – removed some ugly looking template wrappers

Known bugs: none

Comments: I tried to formulate a generic test harness for use by all involved in the cphstl project,
but realized that it was too big a mouthful for now and settled for the below.

*/

long int visits = 0;
#define visit(n) visits +=n

#include <cstdlib>

#include <iostream>
#include <fstream>
#include <time.h> // clock
#include <math.h> // pow
#include <vector>
#include <slist>
#include <functional> // less
#include <string>
#include <iomanip>
#include <cassert>
#include <list>
#include <algorithm>
#include "functions.h"
#include "lazy_select.cpp"

using namespace std;

const long INPUT_SIZE = static_cast<long>( 1000000 );
const long MIN_SIZE = 0;
const long STEP = 1000000;
const unsigned int FIELD_WIDTH = 10;
const unsigned int FUNCTIONNAME_FIELD_WIDTH = 30;
const unsigned int CONTAINERNAME_FIELD_WIDTH = 15;
const unsigned int NUM_ITERATIONS = 3;

template <class C>
void bench( C container,
           string container_name,
           void (*function)( typename C::iterator,
                             typename C::iterator,
                             typename C::iterator ),
           string function_name)
{
    typedef typename C::iterator Iterator;

    clock_t start, stop, acc_time;
    double average_time_in_sec;

```

```

srand( INPUT_SIZE );
ofstream output_file( (container_name + function_name).c_str(), ios::trunc | ios::app );
for ( int i = MIN_SIZE; i <= INPUT_SIZE; i += STEP )
{
    acc_time = 0;
    visits = 0;
    for ( unsigned int j = 0; j < NUM_ITERATIONS; j++ )
    {
        Iterator input_end = container.begin();
        std::advance( input_end, i == 0 ? 1 : i ); // advance one past the end
        std::generate( container.begin(), input_end, rand );
        Iterator input_median = container.begin();
        std::advance( input_median, i/2 );
        start = clock();
        function( container.begin(), input_median, input_end );
        stop = clock();
        assert( benchmark_functions::partitioned( container.begin(), input_median, input_end ) );
        acc_time += stop - start;
    }
    average_time_in_sec = (double)(acc_time / NUM_ITERATIONS) / CLOCKS_PER_SEC;
    output_file << i << "\t " << visits << endl;

    cout.setf( ios::left );
    cout << " Container : " << setw( CONTAINERNAME_FIELD_WIDTH ) << container_name << "\t"
        << " Function : " << setw( FUNCTIONNAME_FIELD_WIDTH ) << function_name << "\t" << endl
        << " Input size ( ints ) : " << setw( FIELD_WIDTH ) << i << "\t"
        << " Visits : " << setw( FIELD_WIDTH ) << visits << endl << endl;
}
output_file.close();
}

template <class C>
void bench_sort( C container,
                string container_name )
{
    typedef typename C::iterator Iterator ;

    clock_t start, stop, acc_time;
    double average_time_in_sec;

    ofstream output_file( (container_name + "sort").c_str(), ios::trunc | ios::app );
    for ( int i = MIN_SIZE; i <= INPUT_SIZE; i += STEP )
    {
        acc_time = 0;
        visits = 0;
        for ( unsigned int j = 0; j < NUM_ITERATIONS; j++ )
        {
            Iterator end_iterator = container.begin();
            std::advance( end_iterator, i );
            std::generate( container.begin(), end_iterator, rand );
            start = clock();
            container.sort();
            stop = clock();
            acc_time += stop - start;
        }
        average_time_in_sec = (double)(acc_time / NUM_ITERATIONS) / CLOCKS_PER_SEC;
    }
}

```

```

    output_file << i << "\t" << visits << endl;

    cout.setf( ios::left );
    cout << " Container : " << setw( CONTAINERNAME_FIELD_WIDTH ) << container_name << "\t"
        << " Function : " << setw( FUNCTIONNAME_FIELD_WIDTH ) << "sort" << "\t" << endl
        << " Input size ( ints ) : " << setw( FIELD_WIDTH ) << i << "\t"
        << " Visits : " << setw( FIELD_WIDTH ) << visits << endl << endl;
    }
    output_file.close();
}

// these wrapper functions proved necessary to circumvent an internal error in the gcc compiler.

template <class ForwardIterator>
void inline trivial_nth_element_optimized( ForwardIterator first, ForwardIterator nth, ForwardIterator
    last )
{
    benchmark_functions::trivial_nth_element_optimized( first, nth, last );
}

template <class ForwardIterator>
void inline trivial_nth_element_plain( ForwardIterator first, ForwardIterator nth, ForwardIterator last )
{
    benchmark_functions::trivial_nth_element( first, nth, last );
}

template <class Iterator>
void inline select16( Iterator first , Iterator nth, Iterator last )
{
    benchmark_functions::select<Iterator, 16>( first , nth, last );
}

template <class Iterator>
void inline select16_without_pq_distance_optimization( Iterator first, Iterator nth, Iterator last )
{
    benchmark_functions::select_without_pq_distance_optimization<Iterator, 16>( first, nth, last );
}

template <class Iterator>
void inline select_with_unrolled_median( Iterator first, Iterator nth, Iterator last )
{
    benchmark_functions::select_with_unrolled_median<Iterator>( first, nth, last, less< std::iterator_traits
        <Iterator>::value_type>() );
}

template <class Iterator>
void inline random_sampling_with_select( Iterator first, Iterator nth, Iterator last )
{
    benchmark_functions::random_sampling_with_select( first, nth, last, less< std::iterator_traits<Iterator
        >::value_type>() );
}

template <class Iterator>
void inline random_sampling_with_optimized_select( Iterator first, Iterator nth, Iterator last )
{
    benchmark_functions::random_sampling_with_optimized_select( first, nth, last, less< std::

```

```

        iterator_traits<Iterator>::value_type>() );
    }

template <class Iterator>
void inline random_sampling_with_randed_select( Iterator first, Iterator nth, Iterator last )
{
    benchmark_functions::random_sampling_with_randed_select( first, nth, last, less< std::iterator_traits
        <Iterator>::value_type>() );
}

template <class Iterator>
void inline jesper_random_sampling( Iterator first, Iterator nth, Iterator last )
{
    Reengineered_Layerwise::nth_element( first, nth, last , less< std::iterator_traits <Iterator>::value_type
        >() );
}

template <class Iterator>
void inline lazy_select( Iterator first , Iterator nth, Iterator last )
{
    lazy_select( first , nth, last , less< std::iterator_traits <Iterator>::value_type>() );
}

template <class Iterator>
void inline random_sampling_with_nth_element( Iterator first, Iterator nth, Iterator last )
{
    benchmark_functions::random_sampling_with_nth_element( first, nth, last, less< std::iterator_traits<
        Iterator>::value_type>() );
}

int main()
{
    vector<int> v( INPUT_SIZE );

    bench( v, "vector_int_", lazy_select, "lazy_select" );
    bench( v, "vector_int_", benchmark_functions::trivial_nth_element_optimized, "
        trivial_nth_element_optimized" );
    bench( v, "vector_int_", benchmark_functions::trivial_nth_element_plain, "
        trivial_nth_element_plain" );
    bench( v, "vector_int_", std::nth_element, "std_nth_element" );
    bench( v, "vector_int_", benchmark_functions::randed_select, "randed_select" );
    bench( v, "vector_int_", select16, "select_with_pq_distance_optimization" );
    bench( v, "vector_int_", select16_without_pq_distance_optimization, "
        select_without_pq_distance_optimization" );
    bench( v, "vector_int_", select_with_unrolled_median, "select_with_unrolled_median" );

    list <int> l( INPUT_SIZE );

    bench( l, "list_int_", lazy_select, "lazy_select" );
    bench( l, "list_int_", benchmark_functions::trivial_nth_element_optimized, "
        trivial_nth_element_optimized" );
    bench( l, "list_int_", benchmark_functions::trivial_nth_element_plain, "trivial_nth_element_plain
        " );
    bench( l, "list_int_", benchmark_functions::randed_select, "randed_select" );
    bench( l, "list_int_", select16, "select_with_pq_distance_optimization" );
    bench( l, "list_int_", select16_without_pq_distance_optimization, "

```

```

        select_without_pq_distance_optimization" );
    bench( 1, "list_int_", select_with_unrolled_median, "select_with_unrolled_median" );
    bench_sort( 1, "list_int_" );

    slist <int> sl( INPUT_SIZE );

    bench( sl, "slist_int_", lazy_select, "lazy_select" );
    bench( sl, "slist_int_", benchmark_functions::trivial_nth_element_optimized, "
        trivial_nth_element_optimized" );
    bench( sl, "slist_int_", benchmark_functions::trivial_nth_element_plain, "
        trivial_nth_element_plain" );
    bench( sl, "slist_int_", benchmark_functions::randized_select, "randized_select" );
    bench( sl, "slist_int_", select16, "select_with_pq_distance_optimization" );
    bench( sl, "slist_int_", select16_without_pq_distance_optimization, "
        select_without_pq_distance_optimization" );
    bench( sl, "slist_int_", select_with_unrolled_median, "select_with_unrolled_median" );
    bench_sort( sl, "slist_int_" );

}

```

Listing 7: "Benchmarking harness (main() code) for execution time benchmarking "

/* Description: Harness for running benchmark tests on nth_element implementations.

Author: Lars Yde (larsyde@diku.dk)

Revision history: 28/11/2000 – coding begun

19/12/2000 – removed some ugly looking template wrappers

Known bugs: none

Comments: I tried to formulate a generic test harness for use by all involved in the cphstl project, but realized that it was too big a mouthful for now and settled for the below.

*/

```

long int visits = 0;
#define visit(n) visits +=n

#include <cstdlib>
#include <iostream>
#include <fstream>
#include <time.h> // clock
#include <math.h> // pow
#include <vector>
#include <slist>
#include <functional> // less
#include <string>
#include <iomanip>
#include <cassert>
#include <list>
#include <algorithm>

#include "functions.h"
#include "lazy_select.cpp"

using namespace std;

```

```

const long INPUT_SIZE = static_cast<long>( 5000000 );
const long MIN_SIZE = 0;
const long STEP = 1000000;
const unsigned int FIELD_WIDTH = 10;
const unsigned int FUNCTIONNAME_FIELD_WIDTH = 30;
const unsigned int CONTAINERNAME_FIELD_WIDTH = 15;
const unsigned int NUM_ITERATIONS = 3;

template <class C>
void bench( C container,
           string container_name,
           void (*function)( typename C::iterator,
                             typename C::iterator,
                             typename C::iterator ),
           string function_name)
{
    typedef typename C::iterator Iterator;

    clock_t start, stop, acc_time;
    double average_time_in_sec;

    srand( INPUT_SIZE );
    ofstream output_file( (container_name + function_name).c_str(), ios::trunc | ios::app );
    for ( int i = MIN_SIZE; i <= INPUT_SIZE; i += STEP )
    {
        acc_time = 0;
        for ( unsigned int j = 0; j < NUM_ITERATIONS; j++ )
        {
            Iterator input_end = container.begin();
            std::advance( input_end, i == 0 ? 1 : i ); // advance one past the end
            std::generate( container.begin(), input_end, rand );
            Iterator input_median = container.begin();
            std::advance( input_median, i/2 );
            start = clock();
            function( container.begin(), input_median, input_end );
            stop = clock();
            assert( benchmark_functions::partitioned( container.begin(), input_median, input_end ) );
            acc_time += stop - start;
        }
        average_time_in_sec = (double)(acc_time / NUM_ITERATIONS) / CLOCKS_PER_SEC;
        double time_pr_element = i == 0 ? 0 : ( average_time_in_sec / i ) * pow((double)10,(double)9);
        output_file << i << "\t" << time_pr_element << endl;

        cout.setf( ios::left );
        cout << " Container : " << setw(CONTAINERNAME_FIELD_WIDTH) << container_name << "\t"
             << " Function : " << setw( FUNCTIONNAME_FIELD_WIDTH ) << function_name << "\t" << endl
             << " Input size ( ints ) : " << setw( FIELD_WIDTH ) << i << "\t"
             << " Execution time pr. elem (ns) : " << setw( FIELD_WIDTH ) << time_pr_element << endl <<
             endl;
    }
    output_file.close();
}

template <class C>
void bench_sort( C container,
                string container_name)
{

```

```

typedef typename C::iterator Iterator ;

clock_t start, stop, acc_time;
double average_time_in_sec;

ofstream output_file( (container_name + "sort").c_str(), ios::trunc | ios::app );
for ( int i = MIN_SIZE; i <= INPUT_SIZE; i += STEP )
{
    acc_time = 0;
    for ( unsigned int j = 0; j < NUM_ITERATIONS; j++ )
    {
        Iterator end_iterator = container.begin();
        std::advance( end_iterator, i );
        std::generate( container.begin(), end_iterator, rand );
        start = clock();
        container.sort();
        stop = clock();
        acc_time += stop - start;
    }
    average_time_in_sec = (double)(acc_time / NUM_ITERATIONS) / CLOCKS_PER_SEC;
    double time_pr_element = i == 0 ? 0 : ( average_time_in_sec / i ) * pow( double(10), double(9) );
    output_file << i << "\t " << time_pr_element << endl;

    cout.setf( ios::left );
    cout << " Container : " << setw( CONTAINERNAME_FIELD_WIDTH ) << container_name << "\t"
        << " Function : " << setw( FUNCTIONNAME_FIELD_WIDTH ) << "sort" << "\t" << endl
        << " Input size ( ints ) : " << setw( FIELD_WIDTH ) << i << "\t"
        << " Execution time pr. element (ns) : " << setw( FIELD_WIDTH ) << time_pr_element << endl <<
        endl;
}
output_file.close();
}

// these wrapper functions proved necessary to circumvent an internal error in the gcc compiler.

template <class ForwardIterator>
void inline trivial_nth_element_optimized( ForwardIterator first, ForwardIterator nth, ForwardIterator
    last )
{
    benchmark_functions::trivial_nth_element_optimized( first, nth, last );
}

template <class ForwardIterator>
void inline trivial_nth_element_plain( ForwardIterator first, ForwardIterator nth, ForwardIterator last )
{
    benchmark_functions::trivial_nth_element( first, nth, last );
}

template <class Iterator>
void inline select16( Iterator first, Iterator nth, Iterator last )
{
    benchmark_functions::select<Iterator, 16>( first, nth, last );
}

template <class Iterator>
void inline select16_without_pq_distance_optimization( Iterator first, Iterator nth, Iterator last )

```

```

{
    benchmark_functions::select_without_pq_distance_optimization<Iterator, 16>( first, nth, last );
}

template <class Iterator>
void inline select_with_unrolled_median( Iterator first, Iterator nth, Iterator last )
{
    benchmark_functions::select_with_unrolled_median<Iterator>( first, nth, last, less< std::iterator_traits
        <Iterator>::value_type>() );
}

template <class Iterator>
void inline lazy_select( Iterator first , Iterator nth, Iterator last )
{
    #ifdef DEBUG
        output « " calling " « endl;
    #endif
    lazy_select( first , nth, last , less< std::iterator_traits<Iterator>::value_type>() );
    //trivial_nth_element_optimized( first, nth, last );
}

int main()
{
    vector<int> v( INPUT_SIZE );

    bench( v, "vector_int_", lazy_select, "lazy_select" );
    bench( v, "vector<int>", benchmark_functions::trivial_nth_element_optimized, "
        trivial_nth_element_optimized" );
    bench( v, "vector<int>", benchmark_functions::trivial_nth_element_plain, "
        trivial_nth_element_plain" );
    bench( v, "vector<int>", std::nth_element, "std_nth_element" );
    bench( v, "vector<int>", benchmark_functions::randized_select, "randized_select" );
    bench( v, "vector<int>", select16, "select_with_pq_distance_optimization" );
    bench( v, "vector<int>", select16_without_pq_distance_optimization, "
        select_without_pq_distance_optimization" );
    bench( v, "vector<int>", select_with_unrolled_median, "select_with_unrolled_median" );

    list<int> l( INPUT_SIZE );

    bench( l, "list_int_", lazy_select, "lazy_select" );
    bench( l, "list<int>", benchmark_functions::trivial_nth_element_optimized, "
        trivial_nth_element_optimized" );
    bench( l, "list<int>", benchmark_functions::trivial_nth_element_plain, "trivial_nth_element_plain
        " );
    bench( l, "list<int>", benchmark_functions::randized_select, "randized_select" );
    bench( l, "list<int>", select16, "select_with_pq_distance_optimization" );
    bench( l, "list<int>", select16_without_pq_distance_optimization, "
        select_without_pq_distance_optimization" );
    bench( l, "list<int>", select_with_unrolled_median, "select_with_unrolled_median" );
    bench_sort( l, "list<int>" );

    slist<int> sl( INPUT_SIZE );

    bench( sl, "list_int_", lazy_select, "lazy_select" );
    bench( sl, "slist<int>", benchmark_functions::trivial_nth_element_optimized, "
        trivial_nth_element_optimized" );
    bench( sl, "slist<int>", benchmark_functions::trivial_nth_element_plain, "

```

```

        trivial_nth_element_plain" );
    bench( sl, " slist <int>", benchmark_functions::randized_select, "randized_select" );
    bench( sl, " slist <int>", select16, "select_with_pq_distance_optimization" );
    bench( sl, " slist <int>", select16_without_pq_distance_optimization, "
        select_without_pq_distance_optimization" );
    bench( sl, " slist <int>", select_with_unrolled_median, "select_with_unrolled_median" );
    bench_sort( sl, " slist <int>" );
}

```

Listing 8: "Miscellaneous functions"

```

#include <iterator>           // defines std::iterator_traits
#include <utility>           // defines std::pair
#include <cmath>             // defines the usual mathematical functions
// #include "partition.cpp" // defines the 3 way partitioning routine.
#include <stdlib.h>
#include <algorithm>
#include <functional>

using namespace std;

namespace benchmark_functions
{
    using std::pair;
    // cache block size
    const unsigned int BLOCK_SIZE = 16;
    // the threshold for using random sampling
    const long INPUT_SIZE_THRESHOLD = 1000;
    // the size of the random sample
    const long SAMPLE_SIZE_THRESHOLD = 100;
    // the expected ratio between a sampled sequence and the fixed proportion sample drawn from it
    const long SAMPLE_PROPORTION = 32;

    #include "median5.cpp"
    #include "trivial.cpp"
    #include "select.cpp"
    #include "aux_funcs.cpp"
    #include "randized_select.cpp"
}

```

Listing 9: "Header for miscellaneous functions"

```

#ifndef BENCHMARK_FUNCTIONS
#define BENCHMARK_FUNCTIONS

std::ofstream output( "dump.txt", std::ios::ate );

// #define DEBUG

#include <iterator>
#include <functional>
#include <utility>

using namespace std;

namespace benchmark_functions

```

```

{

template < class item > inline bool partitioned ( item p, item q, item r );

template < class position >
    void inline rotate3( position a, position b, position c );

template < class item, class ordering >
    bool sorted( item p, item r, ordering less );

template < class item >
    inline bool sorted( item p, item r );

template < class position >
    inline std:: iterator_traits < position >:: difference_type distance( position p, position q, std::
        forward_iterator_tag );

template < class position >
    inline std:: iterator_traits < position >:: difference_type distance( position p, position q, std::
        bidirectional_iterator_tag );

template < class position >
    inline std:: iterator_traits < position >:: difference_type distance( position p, position q, std::
        random_access_iterator_tag );

template < class position >
    inline std:: iterator_traits < position >:: difference_type distance( position p, position q );

template < class item, class ordering >
    void insertion_sort( item p, item r, ordering less );

template < class item >
    inline void insertion_sort( item p, item r );

template < class item, class ordering >
    pair< item, item > partition( item p, item q, item r, ordering less, std:: forward_iterator_tag );

template < class item, class ordering >
    pair< item, item > partition( item p, item q, item r,
        ordering less, std:: random_access_iterator_tag );

template < class item >
    inline pair< item, item > partition( item p, item q, item r, std:: bidirectional_iterator_tag );

template < class item >
    inline pair< item, item > partition( item p, item q, item r );

template < class item, class ordering >
    inline pair< item, item > partition( item p, item q, item r, ordering less );

```

```

template <class item, class ordering, std::iterator_traits<item>::difference_type gs>
void select (item p, item q, item r, ordering less);

template < class item, class ordering,
std::iterator_traits < item >::difference_type gs > void select_without_pq_distance_optimization (
    item p,
                                                                    item
                                                                    q
                                                                    ,
                                                                    item
                                                                    r
                                                                    ,
                                                                    ordering
                                                                    less )
                                                                    ;

// select with block size 5 and specialized sort
template < class item, class ordering > void select_with_unrolled_sort (item p,
                                                                    item q,
                                                                    item r,
                                                                    std::iterator_traits < item >::
                                                                    difference_type p_q_distance,
                                                                    ordering
                                                                    less );

template < class item, class ordering > void select_with_unrolled_sort (item p,
                                                                    item q,
                                                                    item r,
                                                                    ordering
                                                                    less );

template <class item, size_t gs>
inline void select (item p, item q, item r);

template <class position, class element, class ordering>
pair<position, position> inline partition( position first , position beyond,
element pivot1, element pivot2,
ordering less , std::forward_iterator_tag);

template <class position, class element, class ordering>
pair<position, position> inline partition(position first , position beyond,
element pivot1, element pivot2,
ordering less , std::random_access_iterator_tag);

template <class position, class element, class ordering>

```

```

pair<position, position> inline partition( position first , position beyond,
                                         element pivot1, element pivot2,
                                         ordering less );

template <class position>
void random_sample(position first, position beyond,
                  long samplesz, std::random_access_iterator_tag tag);

template <class position>
void random_sample(position first, position beyond,
                  std::iterator_traits <position>::difference_type samplesz);

template <class position, class ordering>
void random_sampling_with_select(position first,
                                position nth,
                                position beyond,
                                ordering less );

template <class position, class ordering>
void random_sampling_with_optimized_select(position first,
                                           position nth,
                                           position beyond,
                                           ordering less );

template <class position, class ordering>
void random_sampling_with_randized_select(position first,
                                          position nth,
                                          position beyond,
                                          ordering less );

template <class item>
item pivot(item p, item r);

template <class item, class ordering>
item partition_rselect ( item p, item q, item r, ordering less , std::forward_iterator_tag );

template <class item, class ordering>
item partition_rselect (item p, item q, item r, ordering less , std::random_access_iterator_tag );

template <class item, class ordering>
inline item partition_rselect ( item p, item q, item r, ordering less );

template <class item, class ordering>
void randized_select(item p, item q, item r, ordering less );

template <class item>
inline void randized_select(item p, item q, item r);

template <class position, class ordering>
void randomized_median_of_3_FIND(position first, position nth,
                                position beyond, ordering less );

```

```

template < class ForwardIterator >
    void trivial_nth_element( ForwardIterator, ForwardIterator, ForwardIterator );
}
#include "functions.cpp"

#endif

```

Listing 10: "Median-of-5 macro"

```

/*
 * (C) Lars Yde and Jyrki Katajainen, 2001
 */

#include <functional>
#include <iostream>
#include <algorithm>
#include <assert.h>
#include <time.h>
#include <iterator>
#include <stdlib.h>
#include <stdio.h>

// median5 finds the median of five elements without iteration

#define OUT(median, to1) {\
    *to1 = median;\
}

#define B(pair1min, pair1max, pair2min, pair2max, un5, to1, less) {\
    if ( less ( pair2max, pair1max ) )\
        if ( less ( un5, pair1min ) )\
            if ( less ( pair2max, pair1min ) )\
                if ( less ( pair2max, un5 ) )\
                    OUT( un5, to1 )\
                else\
                    OUT( pair2max, to1 )\
            else\
                if ( less ( pair1min, pair2min ) )\
                    OUT( pair2min, to1 )\
                else\
                    OUT( pair1min, to1 )\
        else\
            if ( less ( pair2max, pair1min ) )\
                OUT( pair1min, to1 )\
            else\
                OUT( pair2max, to1 )\
    else\
        if ( less ( un5, pair2min ) )\
            if ( less ( pair1max, pair2min ) )\
                if ( less ( pair1max, un5 ) )\
                    OUT( un5, to1 )\
                else\
                    OUT( pair1max, to1 )\
            else\
                OUT( pair1max, to1 )\
        else\

```

```

        if ( less ( pair1min, pair2min ) )\
            OUT( pair2min, to1 )\
        else\
            OUT( pair1min, to1 )\
    else\
        if ( less ( pair1max, pair2min ) )\
            OUT( pair2min, to1 )\
        else\
            OUT( pair1max, to1 )\
}

#define A(pair1min, pair1max, un3, un4, un5, to1, less) {\
    if ( less (un3, un4))\
        B(pair1min, pair1max, un3, un4, un5, to1, less)\
    else\
        B(pair1min, pair1max, un4, un3, un5, to1, less)\
}

#define median5(un1, un2, un3, un4, un5, to1, less) {\
    if ( less (un1, un2))\
        A(un1, un2, un3, un4, un5, to1, less)\
    else\
        A(un2, un1, un3, un4, un5, to1, less)\
}

// used this instead of less<T> because of a djgpp compile error
template <class T>
bool comp( const T& l, const T& r )
{
    return l < r;
}

int main()
{
    unsigned char minArray[] = {1,2,3,4,5};
    unsigned long minArraySize = sizeof( minArray )/ sizeof ( minArray[0] );

    unsigned long count = 0;
    do
    {
        unsigned char median;
        unsigned char minCopy[minArraySize];

        copy( minArray, minArray+minArraySize, minCopy );
        median5( minArray[0], minArray[1], minArray[2], minArray[3], minArray[4], &median, comp<
            unsigned char> );
        // bubble sort input array (had problems using sort with less<unsigned char> under the djgpp
        compiler)
        for ( int a = 0; a < 5; a++ )
        {
            for ( int b = 0; b < 4; b++ )
            {
                if ( minCopy[ b ] > minCopy[ b+1 ] )
                {
                    unsigned char temp = minCopy[ b+1 ];
                    minCopy[ b+1 ] = minCopy[ b ];
                    minCopy[ b ] = temp;
                }
            }
        }
    }
}

```

```

        }
    }
}
// dump array to stdout
char buffer [200];
sprintf ( buffer , "Test #%%i went OK: %%i %%i %%i %%i %%i, %%i", count, minArray[0], minArray[1],
    minArray[2], minArray[3], minArray[4], median );
std::cout << buffer << endl;
assert ( minCopy[2] == median );
count++;
} while ( next_permutation( minArray, minArray+4 ));
}

```