Pattern-defeating Quicksort

Orson Peters [1]

[1] Leiden University, The Netherlands
orsonpeters@gmail.com

March 19, 2017

Work in progress. Please do not cite.

Years of study have shown quicksort to be the preferred generic in-memory sort. An asymptotically optimal average case runtime of $\Theta(n \log n)$ with great locality of reference and small memory overhead. The drawback is a glaring worst case runtime of $\Theta(n^2)$. This has lead the best sorting algorithms towards hybrid approaches, combining the fast average runtime of quicksort with other properties, such as the guaranteed $\Theta(n \log n)$ worst case of heapsort. In this paper a novel hybrid sort is introduced that aims to improve the runtime of quicksort on certain patterns conjectured to be common in real world data, giving a new best-case runtime of $\Theta(n)$. Empirical study shows that for many inputs pattern-defeating quicksort is faster, and is never significantly slower.

Introduction

Arguably the most used hybrid sorting algorithm at the time of writing is introsort[2]. A combination of insertion sort, heapsort[3] and quicksort[4], it is very fast and can be seen as a truly hybrid algorithm. The algorithm performs introspection and decides when to change strategy using some very simple heuristics. If the recursion depth becomes to deep, it switches to heapsort, and if the partition size becomes too small it switches to insertion sort.

[2] David Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27:983–993, 1997

[3] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964

[4] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962

The goal of pattern-defeating quicksort (or *pdqsort*) is to improve on introsort's heuristics to create a hybrid sorting algorithm with several desirable properties. It maintains quicksort's constant[5] memory usage and fast average case, effectively recognizes and combats worst case behavior (deterministically), and runs in linear time for a few common patterns. It also unavoidably inherits quicksort's instability, so pdqsort can not be used in situations where stability is needed.

[5] Quicksort and pdqsort use $\log n$ memory to maintain the call stack, but this is effectively constant for any practical size.

I have created a state of the art C++ implementation, and explore the design techniques used to achieve a practical high speed implementation. The implementation is fully compatible with `std::sort` and is released under a permissive license. Standard library writers are invited to evaluate and adopt the implementation as their generic unstable sorting algorithm.

A faster solution to the Dutch national flag problem

A naive quicksort implementation might trigger the $\Theta(n^2)$ worst case on the all-equal input distribution by placing equal comparing elements in the same partition. A smarter implementation never swaps equal elements, resulting in average case performance as equal elements will be distributed evenly across the partitions. However, an input with many equal comparing elements is rather common, and we can do better. Handling equal elements efficiently requires tripartite partitioning, which is equivalent to Dijkstra's Dutch national flag problem[6].

Faster in this context means that pdqsort augments quicksort with a capability to handle few distinct elements efficiently with **lower overhead** than previous solutions. It does not mean that it runs faster than dedicated solutions to the Dutch nation flag problem.

[6] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997

Pattern-defeating quicksort uses the fast 'approaching pointers' method[7]. Two indices are initialized, i at the start and j at the end of the sequence. i is incremented and j is decremented while maintaining an invariant, and when both invariants are invalidated the elements at the pointers are swapped, restoring the invariant. The algorithm ends when the pointers cross. Implementers must take great care, as this algorithm is conceptually simple, but is very easy to get wrong.

[7] Jon L Bentley and M Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993

Bentley and McIlroy describe an invariant for partitioning that swaps equal elements to the edges of the partition, and swaps them back into the middle after partitioning. This is efficient when there are many equal elements, but has a significant drawback. Every element needs to be explicitly checked for equality to the pivot before swapping, costing another comparison. This happens regardless of whether there are many equal elements, costing performance in the average case.

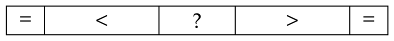


Figure 1: The invariant used by Bentley-McIlroy. After partitioning the equal elements stored at the beginning and at the end are swapped to the middle.

Unlike previous algorithms, pdqsort's partitioning scheme is not self contained. It uses two separate partition functions, one that groups elements equal to the pivot in the left partition (`partition_left`), and one that groups elements equal to the pivot in the right partition (`partition_right`). Note that both partition functions can always be implemented using a single comparison per element as $a < b \Leftrightarrow a \ngeq b$ and $a \nless b \Leftrightarrow a \geq b$.

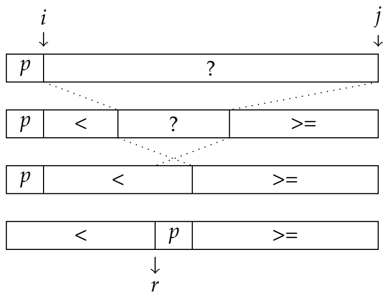


Figure 2: The invariant used by `partition_right` of pdqsort, shown at respectively the initial, halfway and finished state. When the loop is done the pivot gets swapped into its correct position. p is the single pivot element. r is the pointer returned by the partition routine indicating the pivot position. The dotted lines indicate how i and j change as the algorithm progresses. This is a simplified representation, e.g. i is actually off by one.

For brevity I will be using a simplified C++ implementation to illustrate pdqsort. It only supports `int` and compares using comparison operators. It is however trivial to extend this to arbitrary types and custom comparator functions. To pass subsequences around, the C++ convention is used of one pointer at the start, and one pointer at one-past-the-end.

The partition functions assume the pivot is the first element, and that it has been selected as a median of at least three elements in the subsequence. This saves a bound check in the first iteration.

```
int* partition_left(int* l, int* r) {
    int* i = l; int* j = r;

    while (*--j > *l);
    if (j + 1 == r) while (i < j && *++i <= *l);
    else            while (         *++i <= *l);


    while (i < j) {
        std::swap(*i, *j);
        while (*--j > *l);
        while (*++i <= *l);
    }

    std::swap(*l, *j);
    return j;
}
```

```
int* partition_right(int* l, int* r) {
    int* i = l; int* j = r;

    while (*++i < *l);
    if (i - 1 == l) while (i < j && *--j >= *l);
    else            while (         *--j >= *l);

    // bool no_swaps = (i >= j);    (8)
    while (i < j) {
        std::swap(*i, *j);
        while (*++i < *l);
        while (*--j >= *l);
    }

    std::swap(*l, *(i - 1));
    return i - 1;
}
```

[8] A pre-partitioned subsequence will perform no swaps. It's possible to detect this with a single comparison of pointers, no_swaps. This is used for a heuristic later.

Given a subsequence α let us partition it using partition_right:

α

$<$	p	$>=$

$<$	p	q	b

If $p \neq q$ we have $q > p$, and apply partition_right on q, b. Rename q and b as the left partition:

$<$	p	q	b	$>$[9]

[9] Even though partition_right puts equal elements in the right partition, from the perspective of α there will be no equal elements to p in the right partition as $q > p$.

We apply the above step recursively as long as $p \neq q$. If at some point q, b is empty, we can conclude there were no elements equal to p and the tripartite partitioning was done when we initially partitioned α. Otherwise, consider $p = q$. Because of α we know that $\forall x \in b\colon x \geq p$. It is easy to see that $\nexists x \in b\colon x < q$. If we were to partition q, b using partition_left, any element smaller than or equal to q would be partitioned left. However, we just concluded that b can not contain elements smaller than q. Thus, b's left partition only contains elements equal to q, and its right partition only contains elements bigger than q:

$<$	p	$=$	q	$>$

$\blacksquare$

This leads to the partitioning algorithm used by pdqsort. If a subsequence has a *predecessor*[10] p that compares equal to the chosen pivot q, apply partition_left, otherwise apply partition_right. No recursion on the left partition of partition_left is needed, as it contains only equivalent elements.

[10] The element directly preceding it in the original sequence. This is the pivot p for an ancestor α. A subsequence that is *leftmost* has no predecessor.

A linear time best case of pdqsort

Lemma 1. *Any predecessor of a subsequence*[11] *was the pivot of an ancestor.*

[11] Assuming only subsequences passed into recursive calls of pdqsort.

Proof. If the subsequence is a direct right child of its parent partition, its predecessor is obviously the pivot of the parent. However, if the subsequence is the left child of its parent partition, its predecessor is the predecessor of its parent. Since the base subsequence has a predecessor it is not leftmost and there must exist some ancestor of which the subsequence is a right child. ■

I find it important to take a moment and defend the concept of a best-case complexity. People have argued that any algorithm can trivially add a best-case scenario. (A 'cheat' that pdqsort uses as well, for linear time on ascending sequences.) I would like to argue that pdqsort has a 'true' best-case complexity for sequences with many equivalent elements. It converges on linearity smoothly as k goes down, and even if only parts of sequence have many equivalent elements, **recursive calls on those parts will run in linear time**. Finally, I argue that sequences with many equivalent elements are common, and the overhead pdqsort uses to achieve linear time on those sequences is extraordinarily low (a single comparison of the pivot to the predecessor per partition operation).

Lemma 2. *The first time a distinct value v is selected as a pivot, it can't be equal to its predecessor.*

Proof. Assume v is equal to its predecessor. By lemma 1 this predecessor was the pivot of an ancestor partition. This is a contradiction, thus v is not equal to its predecessor. ■

Corollary 2.1. *The first time v is selected as a pivot, it is always used to partition with `partition_right`, and all elements x such that $x = v$ end up in the right partition.*

Lemma 3. *Until an element equal to v is selected as a pivot again, for all $x = v$, x must be in the partition directly to the right of v.*

Proof. By corollary 2.1, x can not be in a partition to the left of v. There also can't be a pivot $p \neq v$ such that $v < p < x$. ■

Lemma 4. *The second time a value equal to v is selected as a pivot, all $x = v$ are in the correct position and are not recursed upon any further.*

Proof. The second time another element $x = v$ is selected as a pivot, lemma 3 shows that v must be its predecessor, and thus it is used to partition with `partition_left`. In this partitioning step all elements equal to x (and thus equal to v) end up in the left partition, and are not further recursed upon. Lemma 3 also provides the conclusion we have just finished processing **all** elements equal to v passed to pdqsort. ■

Theorem 1. *pdqsort has complexity $O(nk)$ when the input distribution has k distinct values.*[12]

[12] Note that this is an upper bound. When k is big $O(n \log n)$ still applies.

Proof. Lemma 4 proves that every distinct value can be selected as a pivot at most twice, after which every element equal to that value is sorted correctly. Each partition operation has complexity $O(n)$. The worst-case runtime is thus $O(nk)$. ■

An optimistic best case

As a note on the previous page I mentioned the *true* best-case of pdqsort, and a 'cheat'. The cheat is specifically aimed at a set of inputs that I argue are vastly overrepresented in the domain of inputs to sorting functions: ascending, descending, and ascending with an arbitrary unsorted element appended. This optimization is due to Howard Hinnant[13]s `std::sort`.

[13] Howard Hinnant et al. libc++ C++ standard library. `http://libcxx.llvm.org/`, 2016. [Online; accessed 2015]

During the computation of `partition_right`, it can be detected if no swaps (a *perfect partition*) were performed using a single pointer comparison operation. If this is detected, and the partition was reasonably balanced[14], we will do a *partial insertion sort* on both entire partitions, regardless of their size.

[14] I will elaborate on reasonably balanced partitions in the worst case section below.

```
bool partial_insertion_sort(int* l, int* r) {
        if (l == r) return true;
        int limit = 0;

        for (int* cur = l + 1; cur != r; ++cur) {
            if (limit > partial_insertion_sort_limit) return false;
            int* sift = cur; int* sift_1 = cur - 1;

            if (*sift < *sift_1) {
                int tmp = *sift;
                do { *sift-- = *sift_1; }
                while (sift != l && tmp < *--sift_1);
                *sift = tmp; limit += cur - sift;
            }
        }

        return true;
}
```

A partial insertion sort works exactly like regular insertion sort, except it keeps track of how many elements it moves. After placing an element in its correct position it will check if the number of moves is less than some limit (a safe estimate I used in my implementation is 8, to make sure there is minimum overhead in the case of a misdiagnosed best case). If not, it will terminate the sort and report the sort failed. If the sort completes successfully this is also reported.

The effect is that small disturbances in a near-sorted array can be fixed and a single element at the end can be moved to its correct position in $O(n)$ time. A descending array will go through one regular partitioning operation, followed by a perfect partition for each half, triggering the partial insertion sort[15], giving a total runtime of $O(n)$ operations.

[15] This is more or less a happy coincidence, as the behavior for a descending array depends on the "approaching pointers" partitioning method pdqsort uses. Furthermore, this also requires the median-of-3 pivot selection to actually sort the values, rather than just selecting the median

The overhead for this optimization is beyond measure, as perfect partitions become exceedingly unlikely for large arrays to happen by chance. A hostile attacker crafting worst-case inputs is also no better off. The maximum overhead per partition is n operations, so it doubles performance at worst, but this can already be done to pdqsort by crafting a worst case that degenerates to heapsort. Additionally, the partial insertion sort is only triggered for a reasonably balanced partition, forcing an attacker to generate good quicksort progress if she intends to trigger repeated misdiagnosed partial insertion sorts.

Maintaining a fast average case

Quicksort has a reputation for a fast average case. However, the difference between an optimized implementation and a naive one can result in a big reduction in runtime. Unless noted otherwise, pattern-defeating quicksort implements a median-of-3 quicksort with all standard optimization techniques.

One of the most well-known tricks is to switch to insertion sort when recursing on a small[16] subarray. But even the insertion sort is subject to optimization:

[16] Benchmarking gave 24 elements as a good general cut-off for insertion sort on various desktop machines. This is the default value in pattern-defeating quicksort, but this value is heavily dependent on cache effects and the cost of a comparison.

```
void insertion_sort(int* l, int* r) {
    if (l == r) return;

    for (int* cur = l + 1; cur != r; ++cur) {
        int* sift = cur; int* sift_1 = cur - 1;

        if (*sift < *sift_1) {
            int tmp = *sift;
            do { *sift-- = *sift_1; }
            while (sift != l && tmp < *--sift_1);
            *sift = tmp;
        }
    }
}
```

This is not the usual way of implementing insertion sort. We don't use swaps, but a series of moves. We move the first check out of the loop to prevent two moves for an element already positioned correctly. And we explicitly use `sift` and `sift_1` to ensure that a compiler will not do needless decrements, even with minimal or no optimization turned on.

A big[17] improvement can be made by eliminating the bounds check in `while (sift != l && tmp < *--sift_1)`. This can be done for any subsequence that is not leftmost, as there must exist an element before `l` that is smaller than or equal to `tmp`, breaking the loop. Although this change is tiny, for optimal performance in a programming language like C (that offers very little metaprogramming) this requires an entirely new function, often called `unguarded_insertion_sort`.

[17] 5-15% from my benchmarks, for sorting small integers.

The majority of the time spent in pattern-defeating quicksort is in `partition_right`. We already listed its implementation above[18], and it's of note that great care was taken to ensure that the inner loop does the least amount of work as possible. Again this is done by not doing bounds checks in the inner loop, instead separately handling the first iteration and using elements from previous iterations as sentinel elements to break the inner loop.

[18] One optimization that is omitted from the listed code above is moving `*l` (the pivot) into a local variable. This is done to prevent the compiler from repeatedly loading it from memory, as it otherwise might not be able to prove that it does not change while partitioning.

During the development of pdqsort I was notified of Edelkamp and Weiß' recent work on BlockQuicksort[19]. Their technique gives a great[20] speedup by eliminating branch predictions during partitioning. In pdqsort it is only applied for `partition_right`, as the code size is significant and `partition_left` is rarely called.

[19] Stefan Edelkamp and Armin Weiß. BlockQuicksort: How branch mispredictions don't affect quicksort. *CoRR*, abs/1604.06697, 2016

[20] 50-80% from my benchmarks, for sorting small integers.

Branch predictions are eliminated by replacing them with data-dependent moves. First some static block size is determined[21]. Then, until there are less than `2*block_size` elements remaining, we repeat the following process.

[21] In my implementation I settled on a static 64 elements, but the optimal number depends on your CPU architecture as well as the data you're sorting.

We look at the first `block_size` elements on the left hand side. If an element in this block is bigger or equal to the pivot, it belongs on the right hand side. If not, it should keep its current position. For each element that needs to be moved we store its offset in `offsets_l`. We do the same for `offsets_r`, except for the last `block_size` elements, and finding elements that are strictly less than the pivot:

```
int num_l = 0;
for (int i = 0; i < block_size; ++i) {
    if (*(l + i) >= pivot) {
        offsets_l[num_l] = i;
        num_l++;
    }
}
```

```
int num_r = 0;
for (int i = 0; i < block_size; ++i) {
    if (*(r - 1 - i) < pivot) {
        offsets_r[num_r] = i + 1;
        num_r++;
    }
}
```

But this still contains branches. So instead we do the following:

```
int num_l = 0;
for (int i = 0; i < block_size; ++i) {
    offsets_l[num_l] = i;
    num_l += *(l + i) >= pivot;
}
```

```
int num_r = 0;
for (int i = 0; i < block_size; ++i) {
    offsets_r[num_r] = i + 1;
    num_r +=*(r - 1 - i) < pivot;
}
```

This contains no branches. Now we can **unconditionally** swap elements from the offset buffers:

```
for (int i = 0; i < std::min(num_l, num_r); ++i) {
    std::iter_swap(l + offsets_l[i], r - offsets_r[i]);
}
```

Notice that we only swap `std::min(num_l, num_r)` elements. This is because we pair each elements on the left with an element that belongs on the right. Any leftover elements are re-used next iteration[22], however it takes a bit of extra code to do so. It is also possible to re-use the last remaining buffer for the final elements to prevent any wasted comparisons, again at the cost of a bit of extra code. For the full implementation and more explanation I invite the reader to check the Github repository, and read Edelkamp and Weiß' original paper[23].

[22] After each iteration at least one offsets buffer is empty. We fill any buffer that is empty.

[23] Stefan Edelkamp and Armin Weiß. BlockQuicksort: How branch mispredictions don't affect quicksort. *CoRR*, abs/1604.06697, 2016

The concept is important here: replacing branches with data-dependent moves followed by unconditional swaps. This eliminates virtually all branches in the sorting code, as long as the comparison function used is branchless. This means in practice that the speedup is limited to integers, floats, small tuples of those or similar. However, it's still a comparison sort. You can give it arbitrarily complicated branchless comparison functions (e.g. `a*c > b-c`) and it will work, but providing real-world speedups rivaling those of radix sorting algorithms.

The trade-off is approximately a 10% loss of performance for branchy comparison functions. However, the C++ implementation is conservative, and by default only uses block based partitioning if the comparison function is `std::less` or similar, and the elements being sorted are integer or float. If a user wishes to get block based partitioning otherwise it needs to be specifically requested.

Preventing the worst case

Pattern-defeating quicksort calls any partition operation which is more unbalanced than p (where p is the percentile of the pivot, e.g. $\frac{1}{2}$ for a perfect partition) a *bad partition*. Initially, it sets a counter to $\log n$. Every time it encounters a bad partition, it decrements the counter before recursing[24]. If at the start of a recursive call the counter is 0 it uses heapsort to sort this subsequence, rather than quicksort.

[24] This counter is maintained separately in every subtree of the call graph - it is not a global to the sort process. Thus, if after the first partition the left partition degenerates in the worst case it does not imply the right partition is also sorted with heapsort.

Lemma 5. *At most $O(n \log n)$ time is spent in pdqsort on bad partitions.*

Proof. Due to the counter ticking down, after $\log n$ levels that contain a bad partition the call tree terminates in heapsort. At each level we may do at most $O(n)$ work, giving a runtime of $O(n \log n)$. ■

Lemma 6. *At most $O(n \log n)$ time is spent in pdqsort on good partitions.*

Proof. Consider a scenario where quicksort's partition operation always puts pn elements in the left partition, and $(1-p)n$ in the right. This consistently forms the worst possible good partition. Its runtime can be described with the following recurrence relation:

$$T(n,p) = n + T(pn,p) + T((1-p)n,p)$$

For any $p \in (0,1)$ the Akra-Bazzi[25] theorem shows $\Theta(T(n,p)) = \Theta(n \log n)$. ■

[25] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998

Theorem 2. *Pattern-defeating quicksort has complexity $O(n \log n)$.*

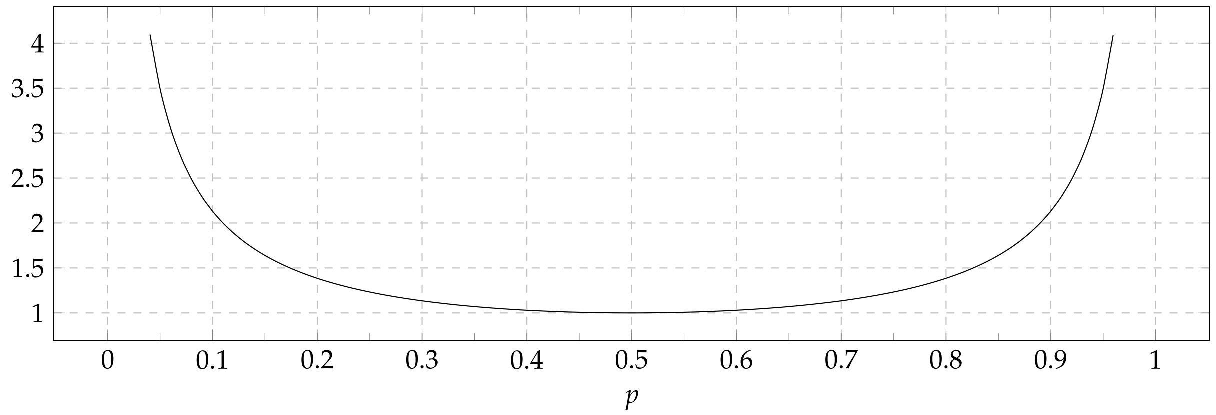
Proof. Pattern-defeating quicksort spends $O(n \log n)$ time on good partitions, bad partitions, and degenerate cases (due to heapsort also being $O(n \log n)$). These three cases exhaustively enumerate any recursive call to pdqsort, thus pattern-defeating quicksort has complexity $O(n \log n)$. ■

We have proven that for any choice of $p \in (0,1)$ the complexity of pattern-defeating quicksort is $O(n \log n)$. However, this does not tell use what a good choice for p is.

From benchmarks I've found that heapsort is approximately twice as slow as quicksort for sorting randomly shuffled data. My goal is to choose p such that $T(n,p)$ is twice as slow as the average $T(n)$. This means that a bad partition becomes synonymous with "worse than heapsort".

The advantage of this scheme is that p can be tweaked if the architecture changes, or you have a different sorting algorithm instead of heapsort.

Slowdown of $T(n,p)$ compared to $T(n,\frac{1}{2})$



The above graph is fundamental[26] in understanding quicksort's performance characteristics. It shows how much slower quicksort becomes compared to the ideal if every partition splits at percentile p. It also beautifully shows why quicksort is generally so fast. Even if every partition is split 80/20, we're still running only 40% slower.

I have chosen $p = 0.125$ as the cutoff value for bad partitions for two reasons: it's reasonably close to being twice as slow as the average sorting operation and it can be computed using a simple bitshift on any platform, because $0.125n = n \gg 3$.

[26] It turns out that:

$$\lim_{n\to\infty} \frac{T(n,p)}{T(n,\frac{1}{2})} = \frac{1}{H(p)}$$

where H is Shannon's binary entropy function:

$$H(p) = -p\log_2(p) - (1-p)\log_2(1-p)$$

Defeating patterns

Quicksort has a love-hate relationship with *patterns*. A pattern is a series of elements that form some self-similar structure after partitioning. This causes a really bad or really good pivot to be repeatedly chosen. We want to eliminate this, as the difference between a good and mediocre pivot is small, but the difference between a mediocre and bad pivot is massive.

A classical way to deal with this is by randomizing pivot selection (also known as randomized quicksort). However, this has multiple disadvantages. Sorting is not deterministic, the access patterns are unpredictable and extra runtime is required to generate random numbers.

Pattern-defeating quicksort takes a different approach. After partitioning we check if the partition was *bad*. If it was, we swap two pairs of elements for each side of the partition. The first element gets swapped with the element at 25% percentile, and the element at 75% percentile gets swapped with the element at the end. Pattern-defeating quicksort chooses the median of the first, middle and last element in a subsequence as the pivot. Since we swap the first and last elements with different elements it introduces fresh candidates for pivot selection.

With this scheme pattern-defeating quicksort is still fully deterministic, has no overhead when the partition is good, and does only a tiny bit of work when the partition was bad. Yet this small shuffle is enough to successfully break up many of the patterns I've tried that regular quicksort struggles with.

Benchmarks

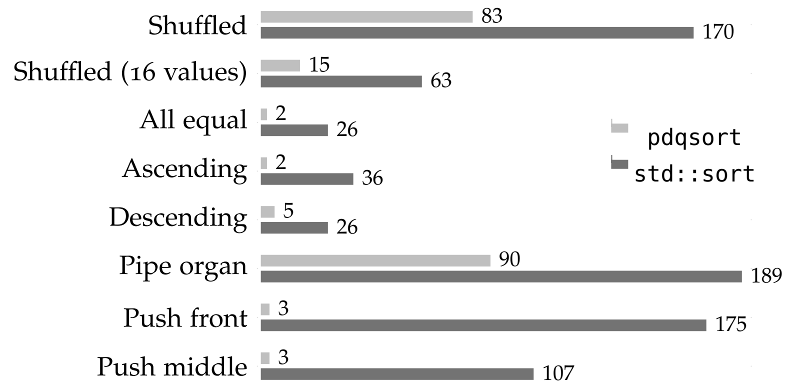
I compared pattern-defeating quicksort's C++ implementation with libstdc++'s `std::sort` implementation. It was **never** significantly[27] slower for the same large input. For smaller input sizes it was slower on one input pattern due to the overhead of block based partitioning. For integers and floats it was significantly faster, as well as for all kinds of patterns.

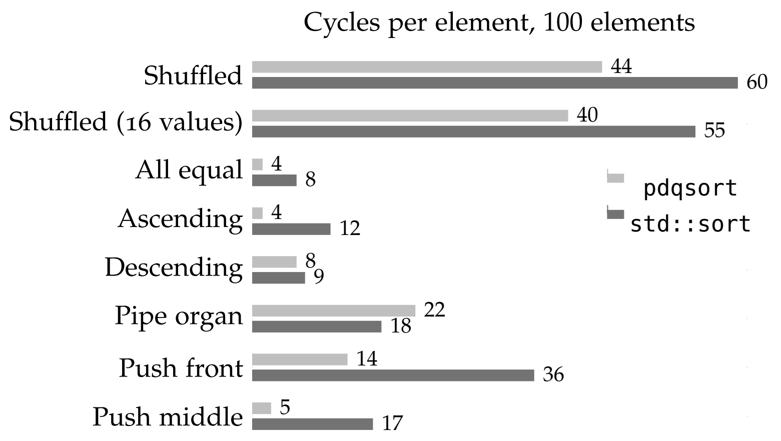
[27] I considered anything within 1% runtime within the measure of error.

For the benchmarks in this paper I focus on the change in performance under various input distributions[28]. All benchmarks were done with integer data, using `g++` 6.3.0 with compilation flags `-std=c++11 -O2 -m64 -march=native` on a Intel i5-4670k @ 3.4GHz.

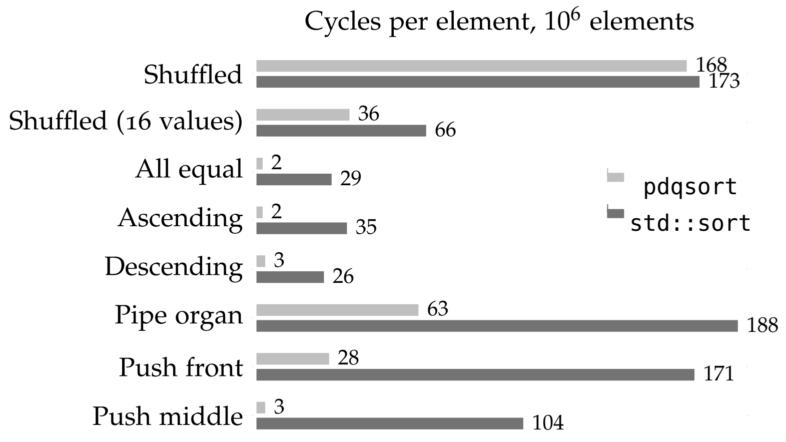
[28] A brief rundown of the distributions used to benchmark. *Shuffled* is simply unique elements in a random order. *Shuffled (16 values)* is similar, but only has 16 distinct values, and thus many equal elements. *All equal* is even more extreme, containing only a single distinct value. *Ascending* and *descending* speak for themselves. *Pipe organ*'s first half is ascending, the second half is descending. *Push front* is an ascending distribution with one element appended to the end that belongs in the front. Finally, *push middle* is similar but the appended element belongs in the middle. I argue that all distributions used are common in real world data with the exception of *pipe organ*, which is used to illustrate defeating a pattern.

Cycles per element, 10^6 elements

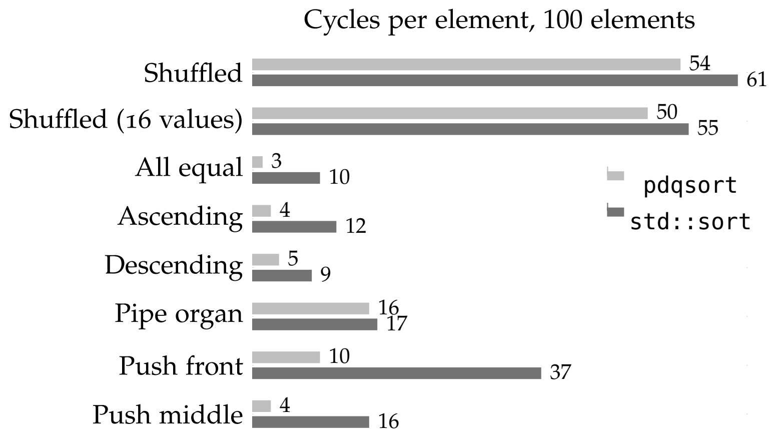




The following two graphs are for pdqsort without block based partitioning. This is the relative performance you can expect for branchy comparison functions, such as for strings or complex user-defined comparison functions.



Note that *pipe organ* is an excellent example of a pattern being defeated. Instead of slightly slowing down like `std::sort`, we actually speed up. The reason `std::sort` slows down on this pattern is due to repeatedly selecting a bad pivot. Pattern-defeating quicksort only selects a bad pivot once, and then speeds up due to the branch predictor having an easier time with the repeated pattern. It's interesting to see that the block based partitioning version above does not speed up, as there is no branch predictor to take advantage of.



References

[1] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.

[2] Jon L Bentley and M Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993.

[3] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[4] Stefan Edelkamp and Armin Weiß. BlockQuicksort: How branch mispredictions don't affect quicksort. *CoRR*, abs/1604.06697, 2016.

[5] Howard Hinnant et al. libc++ C++ standard library. `http://libcxx.llvm.org/`, 2016. [Online; accessed 2015].

[6] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[7] David Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27:983–993, 1997.

[8] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.