File: C:/bsd/rigel/sort/Papers/Hybrid.docx
Date: 2023 July

Hybrid Quicksort for Referenced Items with 1 and 3 pivots

Dennis de Champeaux
Silicon Valley
ddc2@ontooo.com

Abstract

This paper has multiple deliverables: the need for two different quicksort algorithms, the need for a hybrid with more than 2 design layers and at least 3 members, a new technique for obtaining pivots, a novel one pivot version extended with the Edelkamp enhancement and several three pivot versions. Each topic has its own section with a description of the underlying ideas and the benefits shown by performance results. Topic N+1 builds on top of topic N. All versions have O(NlogN) worst case complexity and linear complexity on constant inputs. Novel design elements are, among others: using the single element moving technique (in addition to pairwise swapping) and using array layouts with two gaps (in addition to single gap layouts). Given our focus on arrays with referenced items we assess our versions (in addition to timings) with comparison counts instead of with swap counts. We compare our versions favorably against quicksort versions we found in libraries and elsewhere.

1 Introduction

While quicksort was invented over 60 years ago by T. Hoare, [Hoare], there are still novel findings. Two ingenious versions were developed around 1990 (by Schmidt and Bentley & McIlroy [B&M]) that can be applied to arrays with native types (int, float, etc), with referenced types, and even to arrays with fixed length strings. Quicksort had the reputation of quadratic worst case behavior. D. Musser fixed this in 1997 by switching to heapsort when the recursion depth gets too high thereby guaranteeing O(NlogN) worst case complexity [Musser]. This insight is plausibly the most significant contribution since 1960. [The quicksort wiki still does not contain this result while insertion sort is referred to multiple times and hence hybrid-ness has been acknowledged.]

A key observation is that two different algorithms are required for sorting arrays with native types (int, floats, etc) and arrays with referenced types. Since most research thus far has used arrays with integers and has used swap counts, our findings have a green field flavor. The need for different algorithms has materialized already in Java: versions for native types and a version for objects, which is however not based on quicksort.

Our performances are compared against those by Schmidt [LQ] and by Bentley & McIlroy [B&M]).

We proceed by listing papers we have encountered over the years. Obviously this is not more than a random selection.

1991-2004 Douglas C. Schmidt/ LQ
This quicksort version was written by Douglas C. Schmidt and is part of the GNU C Library. The ability to sort any type requires a parameter that specifies the size in bytes of an element. Insertion sort is called once at the end, which triggers another round of cache misses. Only the median of three elements is used for the construction of the pivot, and insertion sort is called on segments less than 5, which is unusually

low. The swap operation has overhead due to a design error. The pivot is stored at a corner in other designs, but not in LQ. Hence every swap operation must check whether the pivot was moved and if so the reference to the pivot must be updated. A comparison of LQ against qsort/B&M on a 16M array with uniform data yielded 13% more comparisons and 17% more time.

1993 Bentley & McIlroy/ B&M
We found this qsort version in the Cygwin library [B&M]. It reuses design decisions from LQ and requires also a parameter for specifying the length of elements in bytes. The array layout of the B&M version addresses upfront the anomaly of inputs with numerous, if not all, identical elements, because these can lead to a quadratic disaster. The B&M version stores elements equal to the pivot initially in regions at the corners of the segment. The loop moving to the right stores them at the left side and the reverse for the left moving loop. A key design decision in B&M is the minimization of the number of comparisons. This is achieved in both left and right moving loops by checking whether the gap has closed: did the indices cross? This makes these loops more expensive than loops that rely on a sentinel at the other side - through proper initialization. Moving elements equal to the pivots initially to the corners entail the penalty of secondary moves of these corner regions to the middle. {Numerous libraries contain a broken version of qsort/B&M. We have used a fixed version with upgrades.}

1997 Introsort / Musser
This one-pivot Introsort version is available in C++. It is based on the insight by David Musser [Musser] to keep track of the recursion level and switch to heapsort when too high (or too low when counting down). His original paper does *not* contain the partitioning component. He just focused on its wrapper with a counter of the recursive depth for the partitioning function. Implementations found on the web of Introsort are not competitive. Insertion sort is best invoked immediately when the input segment is small because the segment is already in the data cache. The implementations found, however, use the old design where insertion sort is calling once at the end on the top level input, as in LQ. Another factor slowing down these implementations is that insertion sort is called on segment sizes up to 16, while 12 has been shown to be more effective. Constructing pivots from only three elements is another problem.

Strangely enough the LQ and B&M versions were not updated with the Musser defense.

2008 ASPS/ Chen
We found this atypical algorithm in [Chen]. While quicksort works its way down to smaller sub segments until it bottoms out at using insertion sort, ASPS starts with a small sorted segment in the left corner of the initial segment. Iteratively that segment is increased by ‘taking a bite’ from the unsorted segment at the right. The bite is partitioned using the median element of the sorted segment. The two partitions of the bite are moved in the middle of the sorted segment. The two smaller regions, each with a sorted and an unsorted component, are handled recursively. Whether a Musser type of defense is necessary remains unclear. Whether hybridization and/or parallelization are applicable could be explored but is beyond the scope of this study.

2009 & 2017 DPQ
DPQ is a two pivot quicksort version for native types of Java only and, remarkably, coded in Java, [DPQ]. These features make it off limits. Still we use this version to demonstrate our claim that an algorithm for native types performs poorly on referenced types and the other way around.

2013 Kushagra
This paper, [Kushagra], describes a prototype version with 3 pivots and a single gap array for a permutation of N integers. A port of their code, which had omissions, was not competitive.

2015 Muqaddas

We found this version in [Muqaddas]. The array layout of his Triple State QuickSort (TSQ) version is the Dutch Flag with: elements less than the pivot, a gap, elements equal to the pivot, a gap and elements greater than the pivot. The implementation is difficult due to relying on pairwise swapping to move elements around. He reports: "We trade space for performance, at the price of n/2 temporary extra spaces in the worst case." This disqualifies TSQ because quicksort is in-place (with limited additional stack space).

2016 Aumuller

This system is described in [Aumuller]. It tests integer array versions with up to 9(?) pivots with one gap array layouts. They report:

> The analysis shows that the benefits of using multiple pivots with respect to the average comparison count are marginal and these strategies are inferior to simpler strategies such as the well-known median-of-k approach. A substantial part of the partitioning cost is caused by rearranging elements.

These observations are not surprising to us, as discussed in our section for 3-pivot versions. Using a one gap array layout facilitates coding but causes re-relocations of array elements, as they report.

2016 Sebastian Wild thesis

While there is not a single line of code in the thesis, we can agree with:

> A main result is that multiway partitioning can reduce the number of scanned elements significantly, while it does not save many key comparisons …

He appears not to be aware of the findings in [Aumuller] when there are several pivots with a one gap array layout:

> A substantial part of the partitioning cost is caused by rearranging elements.

Hence we doubt his assertion on page 335:

> Nevertheless, I consider it likely that we will eventually see a clever implementation of four- or six-way Quicksort outperform Java's current implementation.

2016 Kurosawa

This paper, [Kurosawa], proposes using medians of medians recursively for obtaining pivots. It goes beyond the ninether described in [B&M]. It did not work for us. A modified version required 13% more comparisons and 14% more time. Perhaps it works when sorting native types. Our new technique replaces the ninether for large segment, as described below.

2016 Edelkamp

This paper, [Edelkamp], proposes to address cache failures by accumulating the locations of elements to be moved to the right in a buffer of limited size and subsequently similarly for the locations of elements to be moved to the left in another buffer. When both buffers are full the elements are swapped in bulk. Porting *their* code was not successful but we implemented this idea in one of our 1-pivot versions, see below.

As mentioned in the Abstract his paper has a list of demarcated topics:

- The need for two quicksort algorithms
- The need for more than two layers with a Dutch Flag version for the 2nd layer
- Pivot selection beyond the ninether
- One pivot version with the Edelkamp enhancement
- Three pivot versions with different design variants

This break down should help because our versions differ also on other design aspects: using in addition to pairwise swapping the single element moving technique and using also different array layouts. Our performance results are obtained by the accumulation of gains through the sequence of design choices.

2 The need for two quicksort algorithms

This section provides the argument that two different algorithms are required for respectively arrays with native type elements and for arrays with referenced type elements. The root cause is that comparison functions are 'free' for native types and arbitrary expensive for referenced types. Hence sorting native types can minimize array operations at the expense of comparisons and the other way around for referenced types. We use the following cross tests:

- Test of Java's DPQ2 against Cut2, one of our C quicksort versions. The port of DPQ2 to C and comparing it to Cut2 with referenced items yielded a timing ratio of 0.91in favor of Cut2 and a comparison count ratio of 0.52 in favor of Cut2.
- The port of Cut2 to Java and comparing it against DPQ2 on the long data type yielded instead the time ratio 0.78 in favor of DPQ2.

It has taken over a decade to confirm this lingering conjecture. [The root cause: DPQ2 can use insertion sort for segment sizes up to 45 because comparisons are 'free' for native types.]

This result explains why we focus on dealing only with referenced items. It explains also why the two ingenious versions LQ and B&M are actually sub optimal for both types of arrays.

Since the comparison operation can be arbitrary expensive for referenced types we use in our testing comparison counts instead of swap counts. Timing remains the primary metric, of course.

3 More than two layers with a Dutch Flag version for the 2nd layer

3.1 The midsize sorter

A typical, classical quicksort version has two layers:

- Check whether the segment is small, say less than 12, and if so delegate the segment to insertion sort, otherwise
- Use the ninether – with up to nine elements – by taking medians in order to fetch a pivot; subsequently the segment is partitioned and recursion and/ or tail iteration is applied to the obtained segments.

Insertion sort is used to avoid the overhead of obtaining a good pivot on small segments. Since we have developed a more elaborate technique for larger segments than the ninether to obtain a pivot, we can apply the same logic:

- Check whether the segment is less than 250, and if so delegate the segment to a sorter that handles segments less than 250, otherwise
- Use the more elaborate technique to obtain a pivot and proceed as above.

This section is devoted to the sorter that handles the segment sizes less than 250. We need to show that this sorter is better than alternatives for this task. We use a variant of the Dutch Flag algorithm that can also be applied with acceptable performance on large segments. It has a somewhat atypical control flow as shown in Figure 1.

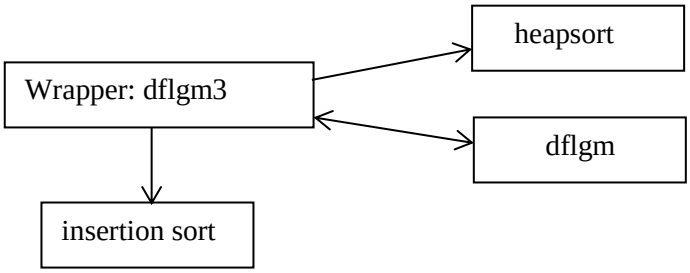

Figure 1. The control flow for a Dutch Flag sorter with *dflgm* for the partitioning, with *dflgm3* for its wrapper, and with the support members insertion sort and heapsort. Notice the cross recursion between *dflgm3* and *dflgm*.

The control flow is atypical because normally the module *dflgm*, which is doing the partitioning, would be inside *dflgm3* so that it can easily execute its recursive invocations. We have split them up because we use *dflgm* also in a different setting.

The signature for *dflgm* (with index encoding), simplified, is:

dflgm(*A, lo, hi, pivotx, cut, depthLimit, compare*)

with *A* the array, *lo* and *hi* the segment boundaries, *pivotx* the index where the pivot is, *cut* the call back function, *depthLimit* for deciding to delegate to heapsort and *compare* the function to compare two elements. The *cut* argument is necessary so that *dflgm* knows inside who the client is.

Unlike the original Dutch Flag algorithm *dflgm* uses a two cap array layout:

< pivot	undecided	= pivot	undecided	> pivot

The implementation of *dflgm* uses the single element moving technique (also known as 'whack a mole') – an alternative to pairwise swapping. We encountered this technique a decade (?) ago in a single gap quicksort version from a Chilean university. The first step is to fetch the left corner element and store it in the variable *x*. The pivot is stored in the corner (which is not strictly required because it will be replaced when partitioning is finished). The idea of this technique (simplified) is captured by the following loop:

- The variable *x* contains a segment element for which it is known to which of the three sub segments it must be added, say the sub segment for '< pivot'
- Copy in *y* the left most element of the left 'undecided' side sub-segment
- Store on that location *x*, set *x = y,* adjust the boundary index and determine where *x* must be inserted

When the undecided gaps are closed the last value in *x* is inserted; either in the left corner or in a location obtained after one or two swaps.

Our two gap array layout aims to decrease re-relocations. The implementation uses a finite state machine with labeled states and transitions with gotos. (Attempts to avoid using gotos have failed – thus far.) There are 4 states with the gaps still open, 3 states with the left gap closed, 3 states with the right gap closed and a final 'done' state. The code of *dflgm* is available at Dsort.c in [GitHub].

The pseudo code of the wrapper *dflgm3* is:

if the segment size is less than 12 delegate the segment to insertion sort;
if the *depthLimit* is too low delegate the segment to heapsort;
decrease *depthLimit*;
set in *px* the pivot index using the ninether;
invoke: *dflgm(A, lo, hi, px, dflgm3, depthLimit, compareXY);*

The code of the wrapper is available at D3sort.c in [GitHub].

3.2 Correctness

The *dflgm* code has a finite state machine in which each label has a specified invariant. The actions in the transitions satisfy these invariants. Termination is guaranteed because the undecided segments shrink. The output being sorted is obtained by the two recursive invocation that produce sorted sub-segments. The output being a permutation of the input is a 2nd order property: the existence of an isomorphism between the array input and output. This proof is involved. The isomorphism maps during partitioning the input array to the current array minus the left corner element and the value in the roving variable *x* (part of the single element moving technique). Inserting *x* and resetting it with *y*, another array element creates the next isomorphism. When the gaps are closed we have an isomorphism involving *x* and the array excluding the left corner. If *x* belong in L it is stored in the left corner and the isomorphism is adjusted. Otherwise one or two swaps are needed to create a gap where *x* can be inserted. The isomorphism is adjusted accordingly.

3.3 Performance of the midsize sorter

Since the Dutch Flag sorter role in our hybrids is taking care of mid-size segments *and* of segments with too many constant elements, we show its versatility as a generic sorter on large 16M segments.
Table 1 has test results for this combination against LQ (qsort made by Schmidt), Bentley (qsort made by Bentley & McIlroy [B&M]).

Version	Comparisons	Timing
LQ	470277970	11.5782
Bentley	415278363	9.8280
Dflgm3	415066913	9.7094

Table 1.The Dutch Flag based sorter Dflgm3 outperforms the quicksort versions LQ and Bentley in the libraries. We used 16M arrays with a uniform distribution of referenced items on an I5 box. The timings are in seconds.

The *Dflgm3* version avoids a quadratic worst case because of the protection provided by the heapsort member and thereby is an improvement over LQ and Bentley. Still a distribution with constant elements could cause excessive recursions. Table 2 has the performance with arrays having constant elements.

Version	Comparisons	Timing
LQ	385876189	2.9062
Bentley	16779922	0.1094
Dflgm3	16782499	0.0624

Table 2. Timings as in Table 1 but using a constant distribution on 16M arrays.

All versions except LQ display linear complexity on constant arrays. The Bentley version obtains this advantage by testing also whether a segment element is equal to the pivot as described above. *Dflgm3/dflgm* obtains the same advantage by maintaining a middle segment from the start.

We use the Dutch Flag sorter in our hybrids primarily for segments in the mid-size range. Table 3 has the data for three different sizes that are representative for this range. The timings and differences are tiny, but this member is executed many times. The *Dflgm3* version has the least number of comparisons and the best timing.

Size	LQ		Bently		Dflgm3	
	Comparisons	Timing	Comparisons	Timing	Comparisons	Timing
128	1002	0.000012	861	0.000010	834	0.000009
256	2308	0.000027	2003	0.000022	1948	0.000020
512	5225	0.000059	4563	0.000049	4454	0.000046

Table 3 Timings as in Table 1 using a uniform distribution.

These performance results show that the Dutch Flag sorter is a good member for our hybrids.

4 Pivot selections beyond the ninether

The ninether uses few resources to obtain a pivot, which is adequate for segments less than 250. For larger segments we can use a different, more extensive technique; see Figure 2 for the subsequent description.

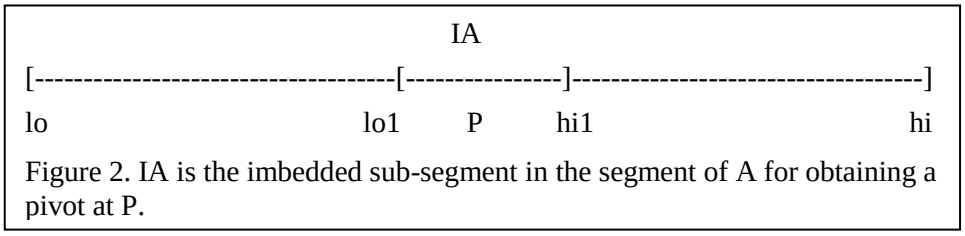

Figure 2. IA is the imbedded sub-segment in the segment of A for obtaining a pivot at P.

The narrative version of this technique:

- Create indices in the middle of the segment, *lo1* and *hi1*, for an imbedded array *IA* in the center that has a size that is proportional to the square root of the size of the segment; *P* is the center index
- Swap equally spaced elements from the segment into *IA*
- Sort *IA* (recursion inside recursion)
- The element at *P* is the candidate pivot
- Check that the elements at the corners of *IA* are different from *P*
- If not (too many equal elements) delegate the segment to the Dutch Flag *dflgm*: provide the identity of the client in *cut*, the pivot index in *P* and with the other arguments and return
- Otherwise swap the left half side of *IA* to the left corner of the *[lo,hi]* segment and similar with the right half side of *IA*
- Partition of the *[lo,hi]* segment using fast loops that do not need bound checks because the corner segments are sentinels

This narrative explains why we have split up the Dutch Flag version in two.
Section 5 shows the performance gains of using this technique versus using the ninether.
Appendix A has the code for this narrative.

4.1 Guarantee linear complexity

While this pivot generator's primary goal is obtaining higher quality pivots the side effect is that it guarantees linear complexity on constant input. Our Cut2 member for large segments is the first version we equipped with this pivot generator. See table 4A in which we compare LQ against Cut2 with constant input on 16M arrays.

Version	Comparisons	Timing
LQ	385875970	2.6625
Cut2	16778780	0.0609
Table 4A. LQ and Cut2 are one pivot sorters with 16M arrays and a constant distribution; the time is again in seconds.		

4.2 Correctness

The only concern regarding using this technique is that array elements are not lost. The creation of the IA segment is done through pairwise swaps. The sorting of IA creates an isomorphism for IA. Delegation to *dflgm* creates another isomorphism. Swapping IA elements to the corners is done through pairwise swaps as well. Hence the technique is safe, while subsequent partitioning proceeds with initialized left and right corners that satisfy the partitioning properties for that segment. Linear complexity for constant arrays is obtained by the delegation to *dflgm*.

5 One pivot version with the Edelkamp enhancement

Our one pivot hybrids have the control flow as shown in Figure 3.

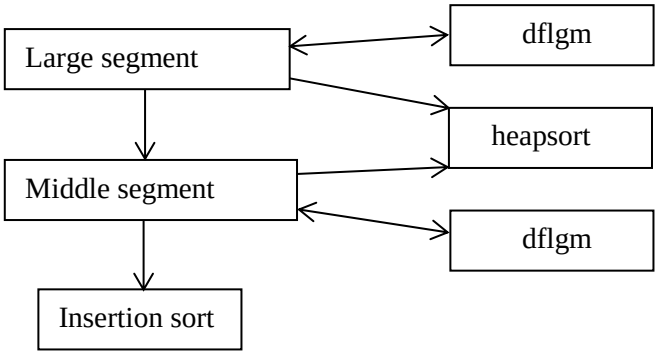

Figure 3. The control flow of our one pivot five member hybrids organized along the size of segments to be processed.

The middle segment sorter is the wrapper *Dflgm3* together with the Dutch Flag module *dflgm* and the two other support members. A large segment sorter delegates mid-size segments down wards. It is a 'genuine' quicksort version with one pivot and uses also heapsort for the ultimate guarantee of O(NlogN) behavior.

5.1 The ninether versus the enhanced technique for obtaining a pivot

We have a capable 1-pivot quicksort version Quick0, which uses the ninether. Another similar 1-pivot version Cut2 is using the alternative technique for obtaining a pivot. Table 4B shows their performances.

Version	Comparisons	Timing
Quick0	415410080	9.768700
Cut2	399599899	9.576600
Table 4B. Quick0 and Cut2 are one pivot sorters; the time is again in seconds.		

Table 4 shows the payoff by using the enhanced technique for obtaining a pivot. The 3.9% decrease of comparisons, now close to the theoretical minimum, has been a pleasant surprise.

The codes of these versions is available on [GitHub] at Qusort.c and C2sort.c on.

5.2 The Edelkamp enhancement

We have another 1-pivot version Cut2lr2 which is like Cut2 but has been extended with the Edelkamp technique. We show a snippet (with index encoding) of what is involved.
Elements to be 'thrown over the wall' are accumulated in two ad hoc buffers and when these are full the elements are swapped in bulk fashion.
The array is: `A`
The pivot is: `T`
The comparison function is: `compareXY`
The size of the two buffers (200) is in: `bufx`
The left side buffer is: `bufl`

Filling the left buffer with an escape when the gap is empty:

```
kl = kr = -1;
while ( kl < bufx ) {
     while ( compareXY(A[++I], T) <= 0 );
     if ( J <= I ) { I = J-1; goto MopUpL; }
     bufl[++kl] = I;
}
```

Bulk swaps are done with:

```
while ( 0 <= kl ) {
   idxl = bufl[kl--];
   idxr = bufr[kr--];
   iswap(idxl, idxr, A);
}
```

A comparison of Cut2 versus Cut2lr2 is in Table 5.

Version	Comparisons	Timing
Cut2	399599899	9.576600
Cut2lr2	400261257	8.765700
Table 5. Cut2 and Cut2lr2 are one pivot sorters; the time is again in seconds.		

We are pleased to show that the Edelkamp technique provides a significant advantage indeed.
The code of Cut2lr2 is available on [GitHub] at C2LR2.c.

5.3 Timings for a range of sizes

We provide in this section performance data for a range of sizes.

Table 6 compares Cut2lr2 against LQ and Bentley on the range 1M – 32M.

Size	LQ		Bently		Cut2lr2	
	Comparisons	Timing	Comparisons	Timing	Comparisons	Timing
1M	24383348	0.4577	21874305	0.3860	21077879	0.3360
2M	51207236	1.0204	46078150	0.8657	44209670	0.7437
4M	107797759	2.3110	96165043	1.9594	92470033	1.6876
8M	225578620	5.2032	200951505	4.4406	192818417	3.8453
16M	468647864	12.0203	416861360	10.1735	400261257	8.7601
32M	969219903	28.7530	860221337	24.5860	825983017	21.6218
Table 6 Timings as in Table 1 using a uniform distribution.						

The timing ratios for Cut2lr2/ Bently on this range is: 0.87, 0.86, 0.86, 0.86, 0.86, 0.88. Increasing the array size to 128M yielded the data in Table 7.

Size	LQ		Bently		Cut2lr2	
	Comparisons	Timing	Comparisons	Timing	Comparisons	Timing
128M	4162554211	110.516	3520257585	90.437	3419149117	78.656
Table 7 Timings as in Table 1 using a uniform distribution.						

The timing ratio for Cut2lr2/ Bently on this range is: 0.87.

6 Three pivot versions with different design variants

This section deals with several 3-pivot sorters having 4 layers. Figure 4 has the control flow in which the 1-pivot sorter stands, for example, for Cut2lr2 with all its other members as shown in Figure 3.

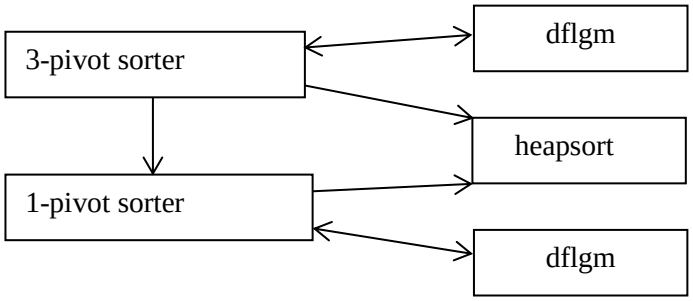

Figure 4. The control flow of our three pivot hybrids organized along the size of the number of pivots.

A three pivot version switches to a 1-pivot sorter for segments that are smaller than, for example, 1500; the switch value depends on the version. A three pivot version can use a similar technique for obtaining its pivots as before. After sorting the mini array in the center of the segment three candidate pivots can be extracted. The test for guaranteeing that sub-segments are disjoint is more elaborate. If they are not disjoint the segment is delegated as before to *dflgm*. Otherwise the 4 sub-segments are positioned in the segment by swapping. Their target locations depend on whether a one gap array layout or a two gap array is used, see Figure 5.

A	B	undecided	C	D

A	undecided	B	C	undecided	D

Figure 5.The A & B regions have elements that are less or equal than then the middle pivot and C & D are larger. A & B and C & D are separated by respectively the left and right pivots.

We developed several 3-pivot versions. They differ regarding the used array layout and other design choices:

- C4: It uses the two gap layout, index encoding, the single element moving technique with a finite state machine
- CN4: It uses the two gap layout, address encoding, the single element moving technique with normal control flow
- CD4: It uses the two gap layout, index encoding, the single element moving technique with normal control flow
- C7: It uses the single gap layout, index encoding and a 'normal' control flow with pairwise swapping

6.1 Correctness

The 3-pivot sorters have similar architectures as the members described above. Hence the same arguments can be used to ascertain their correctness, termination and complexities.

6.2 Timings of three pivot sorters

Table 8 has timings for 3-pivot sorters on 16M arrays. We have added three 1-pivot versions to show the trend.

Version	Comparisons	Timing
Quick0	415410080	9.768700
Cut2	399599899	9.576600
Cut2lr2	400261257	8.765700
Cut4	397408533	8.545400
Cut4n	397423734	7.847000
Cutd	398969276	7.971800
Cut7	397851740	8.079600

Table 8. Quick0, Cut2 and Cut2lr2 are the 1-pivot sorters; the others are 3-pivot sorters. The array size is 16M with uniform distributions; the time is again in seconds.

The rank order based on the timings in Table 8 is 'interesting'. Running this collection of versions on other equipment can change the rank order of the 3-pivot versions. Also, the 1-pivot version Cut2lr2 turned out to be the 'winner' on an AMD desktop machine – but not on the I3 and I5 laptops. Considerations about this observation are below.

The codes for the versions of this section are available in [GitHub] at C4.c, CN4.c, CD4.c, C7.c.

7 Conclusions and Discussion

Sorting of arrays with natives types and with referenced types requires different algorithms to obtain optimal results due to the cost difference of comparison operations. This paper focuses on arrays with referenced items.

A one pivot quicksort has traditionally 2 layers, for insertion sort and for the recursive partitioning module. The segment size determines which layer gets the segment. We have introduced a layer between insertion sort and the layer for large segments. The middle layer uses a sorter based on a variant of the Dutch Flag algorithm (with instead a two gap array layout) and handles segments of size up to 250. While pivots are obtained with the ninether in the middle layer, members for larger segments use a more sophisticated technique for obtaining higher quality pivots. This decreases timings and the comparison count, which gets close to the theoretical minimum.

These enhanced versions outperform the quicksort qsort versions we found in libraries (because these are not dedicated to referenced items and have additional suboptimal designs).

We proceeded with several 3-pivot versions, which employ a 4-layer design. Their comparison counts are also close to the theoretical minimum and their timings are again better than the 1-pivot versions.

Whether future enhancements will require the layer count to go beyond 4 is an intriguing possibility.

Rank orders of our best sorters differ somewhat on the machines we have run tests. Cache hardware differences could be the cause. Optimizations by compilers could be different on different machines. There are additional hardware issues. The pThread package facilitates a design pattern for parallel quicksort (and beyond). Increasing the thread count produces, however, less than expected speedups. Cache congestion is again a likely explanation due to the increased number of locations where the array gets modified. These considerations are, however, beyond the scope of this paper.

Given our timing rank ordering being fluid, we request others to repeat our tests on other hardware. All versions (and more) and drivers are available on [GitHub].

The Aumuller paper, [Aumuller], suggests that multi-pivot quicksort has no future. They are likely correct for array layouts with one gap. Our results show that versions with 3 pivots and with two gap array layouts can improve one pivot versions. We *conjecture/suspect* that 3 pivot versions are the only competitive multi pivot versions.

A port of Java's *timsort* (for objects) to C and comparing it against our *Cut2* one pivot version on a 16M array yielded a comparison count ratio of 0.95 in favor of *timsort*, while the timing ratio is 0.67 in favor of *Cut2*. Avoiding coding sort functionality in Java and using one of our versions appears an advantage for Java users, and possibly for Python users as well.

We found a *qsort* version on my Ubuntu I5 laptop. It is based on merge sort (hence also not in place). The comparison count ratio against *Cut2lr2* is 0.95 in favor of *qsort*, the timing ratio is 0.82 in favor of *Cut2lr2*; against *Cut4d* the ratio is 0.75. Hence using one of our versions appears an advantage for Ubuntu users.

Our 2-pivot versions have not been competitive; see Appendix B for a discussion.

We recommend that the quicksort wiki gets updated with the contribution by D. Musser and hence that the worst case complexity is described as O(NlogN) when heapsort is a member of the hybrids.

8 Miscellaneous

The reported data was obtained on:
-- Intel I5-2410M (I5/w) 2.3 Ghz, Win10+Cygwin, 3 MB Intel® Smart Cache
We tested also on:
-- Intel I5-8256, (I5/u) 1.6 Ghz Ubuntu, 6 MB Intel® Smart Cache

-- AMD fx-8350 (AMD), 4 Ghz, Win8.1+Cygwin, L1 cache 4x64Kb instruction + 8x16Kb data, L2 cache 4x2Mb, L3 cache 8Mb
Data was obtained from runs with 10 iterations and taking the average. The magic numbers were obtained by extensive tests.
Ratios above 1M typically stabilize. Cache issues become more disruptive with larger arrays. Changing hardware has been experimented with two decades ago but has not taken off, see [Tsigas], 2003.

LQ and B&M are examples that use address encoding. DPQ uses, in Java, index encoding. Some of our modules use index encoding and hence would benefit somewhat from address encoding.

Acknowledgement
The Cut4n version was coded by N. Horspool.

References

[Aumuller] Aumuller, M., M. Dietzfelbinger, Pascal Klaue, "How Good is Multi-Pivot Quicksort?", DOI: 10.1145/2963102
ACM Transactions on Algorithms 13(1) October 2015

[B&M] Bentley, J. & M.D. McIlroy, "Engineering a sort function', *Software - Practice and Experience*; Vol. 23 (11), 1249-1265, 1993.

[Chen] Chen, J-C, "Symmetry Partition Sort", *Software-Practice and Experience*, vol 38, no 7, pp 761-773, 2008 June.

[DFwiki] https://en.wikipedia.org/wiki/Dutch_national_flag_problem
(not quoted in the text)

[DPQ] Dual Pivot Quicksort source is at:
http://www.docjar.com/html/api/java/util/DualPivotQuicksort.java.html
And older version of DPQ is described in Vladimir Yaroslavskiy, "Dual-Pivot Quicksort", 2009.
http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf

[Edelkamp] Edelkamp, Stefan; Weiß, Armin (22 April 2016). "BlockQuicksort: How Branch Mispredictions don't affect Quicksort".
https://arxiv.org/abs/1604.06697

[GitHub] https://github.com/ddccc/C7

[Hoare] Hoare, C.A.R., "Quicksort", *Computer Journal*, vol 5, no 1, pp 10-16, 1962.

[Kurosawa] Kurosawa, N., "Quicksort with median of medians is considered practical",
https://arxiv.org/abs/1608.04852

[Kushagra] Kushagra, S., A. Lopez-Ortiz & J. I. Munro, "Multi-Pivot Quicksort: Theory and Experiments", 2013 November,
http://epubs.siam.org/doi/pdf/10.1137/1.9781611973198.6

[LQ] There is no publication of the Doug Schmidt qsort version.

[Muqaddas] Muqaddas, A, "Triple State Quicksort",

https://arxiv.org/abs/1505.00558

[Musser] Musser, D., "Introspective Sorting and Selection Algorithms", *Software: Practice and Experience* (Wiley) **27** (8): 983–993, 1997.

[Sedgewick] Sedgewick, R., "Implementing Quicksort Programs", *CACM*, Vol 21, No 10, Oct. 1978. (not quoted in the text)

[Tsigas] Tsigas, P. & Y. Zhang, "A Simple, Fast Parallel Implementation of Quicksort and its Performance valuation on SUN Enterprise 10000", 2003, available at: http://www.cse.chalmers.se/~tsigas/papers/Pquick.pdf

Appendix A

We show here the code (for the index encoding version of C2LR) for an alternative to the ninether to obtain a pivot for a large segment.

The length of the segment to be partitioned is *L*.

The index to the middle of the segment is *middlex*.

Pair wise swapping is done with *iswap* in which the third argument *A* is the array.

The sorter member is here *cut2lrc1* (part of C2LR) which calls itself on a smaller array.

The small buffers for bulk swapping are *bufl* and *bufr*.

```
int k, lo1, hi1; // for sampling
int probeLng = (int) sqrt(L/6.0);
if ( probeLng < 9 ) probeLng = 9;
int halfSegmentLng = probeLng >> 1; // probeLng/2;
lo1 = middlex - halfSegmentLng; // lo + (L>>1) - halfSegmentLng;
hi1 = lo1 + probeLng - 2;
int offset = L/probeLng;
// assemble the mini array [lo1, hi1]
int yy = lo;
for (k = lo1; k <= hi1; k++)
  { iswap(k, yy, A); yy += offset; }
// sort this mini array to obtain good pivots
cut2lrc1(A, lo1, hi1, bufl, bufr, depthLimit, compareXY);
T = middle = A[middlex]; // pivot
// test pivot against the corners lo1 and hi1
if ( compareXY(A[hi1], middle) <= 0 ||
     compareXY(A[lo1], middle) == 0 ) {
   // escape because we cannot find a good pivot
   // cut2lrc will be applied to two of the three segments produced by dflgm
   dflgm(A, lo, hi, middlex, cut2lrc, depthLimit, compareXY);
   return;
}
// swap the 2 sub segments of [lo1,hi1] to the corners of [lo,hi]
for ( k = lo1; k <= middlex; k++ ) {
  iswap(k, I, A); I++;
}
I--;
for ( k = hi1; middlex < k; k--) {
  iswap(k, J, A); J--;
}
J++;
// partitioning proceeds from here
```

Appendix B

We give an optimistic complexity analysis for 1, 2 and 3 pivot quicksort type sorting algorithms. This is *not* a worst case analysis but a best case analysis, which yields at most conjectures.

We make the following simplifying assumptions:
- We have a zero cost oracle that provides perfect pivots, which yield during partitioning equally sized sub-segments
- We invoke the sorting algorithms all the way down (hence avoiding the improvement of using insertion sort on small segments because we have perfect pivots)
- We assume that elements to be moved do *not* require subsequent re-relocations (as is the case for 1-pivot partitioning)
An array with uniform distribution of integers and with a cheap way to construct pivots approximates these assumptions.
We define:
A as the time to access an array element
S as the time to store an array element
C as the time to compare two elements
T2(N) is the optimal time to sort an array of size *N* with a single pivot
T3(N) is the optimal time to sort an array of size *N* with two pivots
T4(N) is the optimal time to sort an array of size *N* with three pivots
log2, log3 and *log4* are respectively the logarithm functions for base 2, 3 and 4.

Partitioning an array of size *N* into two sub-segments takes the time:
*N * (A + S/2 + C)*
because we must: access each element, store half of them (assuming a perfect pivot), and compare each element against the pivot.
We must do the same for the two equal sub-segments, and since we have a uniform distribution and a perfect pivot the time for the two segments is:
*2 * (N/2) * (A + S/2 + C) =*
*N * (A + S/2 + C)*
We must repeat partitioning *log2(N)* times. Thus the total time for partitioning with one pivot is:
*T2(N) = N*log2(N)*(A + S/2 + C)*

Partitioning an array of size *N* into three sub-segments takes the time:
*N * (A + S*2/3 + C*5/3)*
because we must: access each element, store 2/3 of them (assuming a perfect pivot), and compare each element only once against one pivot with chance 1/3, and with chance 2/3 to check against both pivots; thus:
*1 * (1/3) + 2 * (2/3)* gives the coefficient 5/3.
We must do the same for the three equal sub-segments, and since we have a uniform distribution and perfect pivots the time for the three segments is:
*3 * (N/3) * (A + S*2/3 + C*5/3) =*
*N * (A + S*2/3 + C*5/3)*
We must repeat partitioning *log3(N)* times. Thus the total time for partitioning with two pivots is:
*T3(N) = N*log3(N)*(A + S*2/3 + C*5/3)*
given: *log3(N) = log2(N)*0.6310* we get:*T3(N) = N*log*
*T3(N) = N*log2(N)*(A*0.6310 + S*0.4207 + C*1.051)*

Partitioning an array of size *N* into four sub-segments takes the time:
*N * (A + S*3/4 + C*2*
because we must: access each element, store 3/4 of them (assuming a perfect pivot), compare each element twice, once against the middle pivot and once against the left or right pivot.
We must do the same for the four equal sub-segments, and since we have a uniform distribution and perfect pivots the time for the four segments is:
*4 * (N/4) * (A + S*3/4 + C*2) =*

*N * (A + S*3/4 + C*2)*
We must repeat partitioning *log4(N)* times. Thus the total time for partitioning with three pivots is:
*T4(N) = N*log4(N)*(A + S*3/4 + C*2)*
given: *log4(N) = log2(N)*0.5* we get:
*T4(N) = N*log2(N)*(A*0.5 + S*0.375 + C)*

Lining up T2, T3 and T4 gives:
*T2(N) = N*log2(N)*(A + S/2 + C)*
*T3(N) = N*log2(N)*(A*0.6310 + S*0.4207 + C*1.051)*
*T4(N) = N*log2(N)*(A*0.5 + S*0.375 + C)*
This analysis *suggests*, but does *not* prove, that T4 with 3-pivots has the sweet spot. The next one could be T8 with 7 pivots, but it is unclear how to avoid re-relocations of elements to be moved with that many pivots.

We coded numerous 2-pivot versions. The timings were not competitive and the comparison counts were worse as predicted by the T3(N) equation.

[A first version of this analysis was done around 1984 when teaching at Tulane University. A two pivot two gap version coded in Pascal was rejected at that time and I was scolded by having used gotos.]