Research Article

HIPM Sort: A Heap–Insertion–Pivot–Merge Hybrid Sorting Algorithm and Its Empirical Comparison With Classical Sorting Algorithms

Chandra Shekhar Ram[1,2], Rajnish Kumar Chaturvedi[1], Sumit Dalal[1]

[1]Department of SCSET, Bennett University, UP, India.
[2]Department of Computer Science and Engineering, Sharda School of Computing Science and Engineering, Sharda University, UP, India.

Abstract

Sorting is one of the most frequently used operations in computing, yet no classical algorithm is best fitted for every data size and distribution. This paper proposes HIPM Sort (Heap–Insertion–Pivot–Merge Sort), a hybrid comparison sort that builds a min-heap to impose a partial order, partitions the second half of the array around the element at position 3n/4, sorts each half with insertion sort, and merges the halves. We review bubble, selection, insertion, Shell, merge, heap, quick and Tim sort with their advantages and disadvantages, derive the complexity of HIPM Sort (O(n) best, O(n^2) average and worst, O(n) space), and compare it with all eight algorithms on 100 random arrays of 100 integers. HIPM Sort removes about 74% of the element shifts of insertion sort, matches quick sort at n = 100 (median 3.0 µs versus 3.1 µs), and is 2.0× faster than insertion sort at n = 1000. Its quadratic growth, however, makes it about 3× slower than quick sort at n = 5000, so it is best suited to small and medium arrays. Its recursive version may yield a better result instead of insertion sort while combined with merge sort like in Shell sort and Tim-sort algorithms.

Keywords: sorting algorithms, hybrid algorithm, heap, insertion sort, merge, time complexity, performance evaluation.

I. Introduction

Sorting arranges data in a specific order and underlies searching, database processing, graphics and scientific computing [1]-[3]. Decades of research have produced many comparison-based algorithms whose trade-offs differ in worst-case bound, memory use, stability, adaptivity to presorted input and constant factors [1]-[3]. Simple quadratic methods are competitive on small arrays, whereas O(n log n) methods win as n grows [4]-[6].

This paper proposes and evaluates HIPM Sort (Heap–Insertion–Pivot–Merge Sort). Its contributions are: (i) the hybrid algorithm together with its complexity analysis; (ii) a structured review of bubble, selection, insertion, Shell, merge, heap, quick and Tim sort listing the advantages and disadvantages of each; and (iii) an experimental comparison on 100 random arrays of 100 integers (as the core benchmark), and through an ablation of HIPM's components. Section II surveys related work, Section III reviews the classical algorithms, Section IV presents HIPM Sort, Section V describes the methodology, Section VI reports results, and Section VII concludes.

II. Related Work

Hybridisation is a long-standing engineering practice. Sedgewick showed that quicksort is faster when small subarrays are left to insertion sort [4], an idea adopted in the widely used library sort of Bentley and McIlroy [5]. Introsort falls back to heapsort when quicksort's recursion becomes too deep, which guarantees O(n log n) time [6]. Timsort merges runs found in the data and is the default sort of Python [7] and of OpenJDK for objects [8]; its O(n log n) worst case was proved only later [9]. BlockQuicksort reduces branch mispredictions during partitioning [10], and LaMarca and Ladner showed that cache behaviour can matter as much as instruction counts [11]. Adaptive algorithms that exploit existing order are surveyed in [12], Bender et al. showed that leaving gaps in the array makes insertion sort O(n log n) with high probability [13], and in-place merging trades extra memory for more data movement [14].

HIPM Sort differs from these designs: it does not recurse. Two O(n) preprocessing passes (heap construction and one partitioning pass) exist only to reduce the number of inversions before two insertion sorts, and a single final merge combines the halves.

III. Classical Sorting Algorithms

Bubble sort repeatedly swaps adjacent out-of-order elements until a pass makes no swap [2], [15]. Selection sort repeatedly moves the minimum of the unsorted part to the front [1]. Insertion sort grows a sorted prefix by inserting each

new element and shifting larger ones to the right [2]. Shell sort [16] applies insertion sort to elements h positions apart for a decreasing gap sequence ending in 1; sequences have been studied by Papernov and Stasevich [17], Pratt [18], Sedgewick [19] and Ciura [20]. Merge sort, attributed to von Neumann (1945) [2], sorts both halves recursively and merges them. Heap sort [21] repeatedly extracts the maximum of a binary heap stored in the array, using Floyd's linear-time heap construction [22]. Quick sort [23], [24] partitions around a pivot and recurses on both sides. Tim sort [7] builds ascending runs (using insertion sort for short ones) and merges them under stack invariants that keep the merges balanced.
Table I summarises the advantages and disadvantages of these eight baselines and Table II gives their asymptotic complexity. An algorithm is stable if it preserves the order of equal keys and adaptive if it runs faster on inputs that are already partly sorted [12].

Table I: Advantages and Disadvantages of Classical Sorting Algorithms

Algorithm	Advantages	Disadvantages
Bubble	Very simple; stable; in place; stops after one pass on sorted input (O(n) best case).	$\Theta(n^2)$ comparisons and swaps on average; slowest of the simple methods; little practical value beyond teaching [2], [15].
Selection	At most n−1 swaps (minimum number of writes); in place; time nearly independent of input [1].	$\Theta(n^2)$ comparisons even on sorted input (not adaptive); not stable.
Insertion	Adaptive, O(n + inversions); stable; in place; online; best of the simple methods for small or nearly sorted arrays [2], [12]; the usual base case of hybrids [4], [7].	$\Theta(n^2)$ on average and worst case; many element shifts on large random inputs.
Shell	Sub-quadratic for good gap sequences [16]–[19]; in place; non-recursive; short code.	Running time depends on the gap sequence, and good sequences are found empirically [20]; not stable; analysis of common sequences is incomplete [2].
Merge	Guaranteed $\Theta(n \log n)$; stable; sequential access suits linked lists and external sorting [1], [2].	Needs $\Theta(n)$ auxiliary memory; in-place versions are complex and slower [14]; basic form is not adaptive.
Heap	Guaranteed $O(n \log n)$; in place with O(1) extra space; no bad inputs [21], [22].	Not stable; poor cache locality degrades performance on arrays larger than the cache [11]; not adaptive.
Quick	Fastest general-purpose method on average; in place (O(log n) stack); good cache behaviour [11], [23], [24].	$\Theta(n^2)$ worst case (e.g., sorted input with a fixed pivot) [5]; not stable; sensitive to the pivot rule; needs an introsort fallback for a guarantee [6].
Tim	Adaptive (linear on presorted runs); stable; $O(n \log n)$ worst case [9]; library default [7], [8].	Complex implementation; O(n) extra space; a run-stack invariant bug was found in the OpenJDK version [8].

Table II: Asymptotic Complexity of the Compared Algorithms

Algorithm	Best	Average	Worst	Space	Stable
Bubble (early exit)	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No

Algorithm	Best	Average	Worst	Space	Stable
Insertion	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Shell (Knuth gaps)	O(n log n)	gap-dep.	$O(n^{1.5})$	O(1)	No
Merge	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Heap	O(n log n)	O(n log n)	O(n log n)	O(1)	No
Quick (last pivot)	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	No
Tim	O(n)	O(n log n)	O(n log n)	O(n)	Yes
HIPM (proposed)	**O(n)**	**$O(n^2)$**	**$O(n^2)$**	**O(n)**	No

Complexities follow [1], [2], [25]; Shell sort's worst case is for Knuth's gap sequence [2].

IV. Proposed Algorithm: HIPM Sort

A. Design

HIPM Sort (Algorithm 1, Fig. 1) is named after its four ingredients. Step 1 builds a min-heap over the whole array in O(n) time with Floyd's bottom-up method [22]; the heap property gives the array a partial order and places the minimum at index 0. Step 2 insertion-sorts the first half. Step 3 chooses the element at index 3n/4 as pivot and partitions the second half so that elements not greater than the pivot move to the front. Step 4 insertion-sorts the second half, and Step 5 merges the two sorted halves in O(n) time.

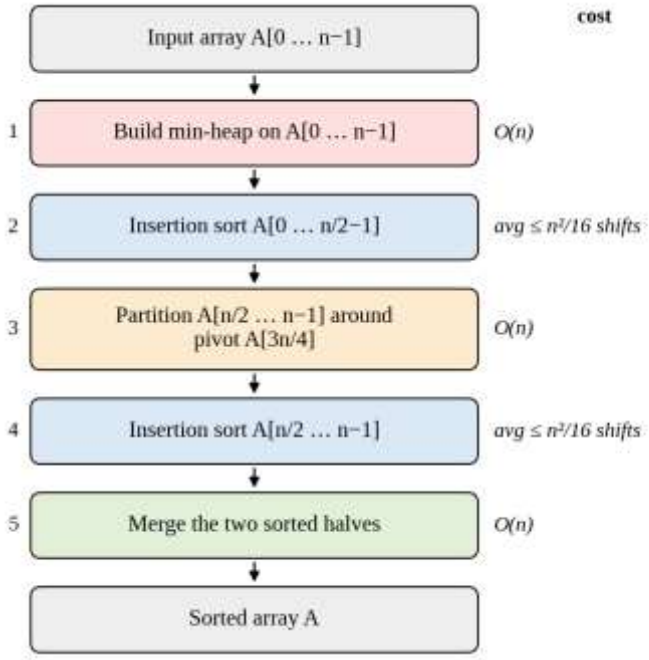


Fig. 1. Flowchart of HIPM Sort with the cost of each stage. Stages 1 and 3 are preprocessing passes; stages 2, 4 and 5 guarantee correctness.

Algorithm 1: HIPM-SORT(A, n)

```
1  BUILD-MIN-HEAP(A, n)
2  h ← n / 2        /* integer division */
3  INSERTION-SORT(A, 0, h − 1)
4  p ← (3 · n) / 4
5  PLACE-PIVOT(A, h, n − 1, p)  /* partition */
6  INSERTION-SORT(A, h, n − 1)
7  MERGE(A, 0, h − 1, n − 1)
```

B. Worked Example

Fig. 2 traces the algorithm on ten integers. Heap construction moves 5 to the root; sorting the left half gives 5, 10, 12, 34, 56; the pivot is A[7] = 45, and partitioning turns the right half 78, 89, 45, 67, 23 into 45, 23, 78, 67, 89, which cuts the number of inversions in that half from eight to two; the final merge produces the sorted array.

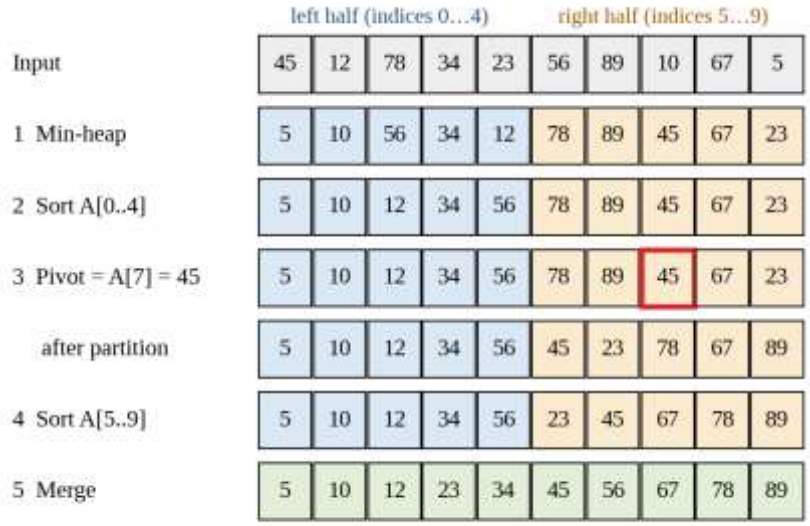


Fig. 2. Trace of HIPM Sort on ten integers. Blue: left half, orange: right half, red frame: pivot, green: merged output.

C. Correctness and Complexity

Correctness. After Steps 2 and 4 each half is sorted regardless of how Steps 1 and 3 permuted the elements, and Step 5 merges two sorted sequences into one; therefore the output is sorted. Steps 1 and 3 change only the cost. Time. Insertion sort of m elements costs $\Theta(m + I)$, where I is the number of inversions [2], and a random permutation has $m(m-1)/4$ inversions. With $m = n/2$ for both halves the expected number of shifts is about $n^2/8$, half of the $n^2/4$ of plain insertion sort; heap ordering and partitioning lower it further (Section VI-D). In the worst case the second half (the leaves of the heap) can still hold $\Theta(n^2)$ inversions after partitioning, e.g., when the elements on each side of the pivot are in decreasing order, so $T(n) = \Theta(n^2)$. For sorted input the heap build, the partition and both insertion sorts leave every element in place, so the best case is $O(n)$. Space and stability. The merge needs $O(n)$ auxiliary space and heap construction uses $O(\log n)$ recursion; the algorithm is not stable because heap construction and partitioning swap equal keys over long distances.

V. Experimental Methodology

All algorithms were written in C and compiled with gcc 13.3 (-O2) on a single-core Linux x86-64 virtual machine (Intel Xeon @ 2.1 GHz). HIPM Sort uses the logic of the proposed program unchanged (the merge buffers are preallocated instead of stack arrays). Bubble sort stops when a pass makes no swap; Shell sort uses Knuth's gaps 1, 4, 13, …; quick sort uses the

Lomuto partition with the last element as pivot and no small-array cutoff; merge sort is top-down with one auxiliary buffer; heap sort uses a max-heap; and Tim sort is simplified: runs of length minrun (computed as in [7]) are insertion-sorted and merged bottom-up, without run detection or galloping. Four experiments were run.

- **E1 (core benchmark):** 100 independent trials, each on a new random array of 100 integers in [0, 9999] (seed 2026); every algorithm sorts the same 100 arrays.
- **E2 (scaling):** n ∈ {50, 100, 200, 500, 1000, 2000, 5000}, 100 random arrays per size.
- **E3 (distributions):** n = 1000 with random, sorted, reversed, nearly sorted (n/20 random swaps applied to a sorted array) and few-unique (10 distinct values) inputs, 100 arrays each.
- **E4 (ablation):** HIPM Sort without the heap step, without the partition step, and without both, compared with plain insertion sort at n = 100 and 1000; element shifts were counted in a separate instrumented build.

One sort call was timed with the POSIX monotonic clock, excluding the array copy; 30 untimed warm-up sorts on unrelated arrays precede each series, and every output was checked against qsort() (all results were correct). We report the mean, median and standard deviation; medians are preferred because isolated operating-system delays create large outliers.

VI. Results and Discussion

A. Core Benchmark: 100 Arrays of 100 Elements

Table III: Time per Sort in Microseconds (n = 100, 100 Runs)

Algorithm	Mean	Median	Std.	Min	Max
Bubble	27.72	27.10	4.33	22.51	58.60
Selection	14.18	13.74	2.61	12.94	34.25
Insertion	2.73	2.70	0.39	2.31	6.35
Shell	3.75	3.71	0.45	3.30	7.99
Merge	4.63	4.23	3.63	3.96	40.49
Heap	3.45	3.41	0.40	3.03	7.13
Quick	3.09	3.05	0.34	2.82	6.29
Tim	2.79	2.57	2.14	2.31	23.95
HIPM	**3.19**	**3.03**	**1.24**	**2.74**	**14.71**

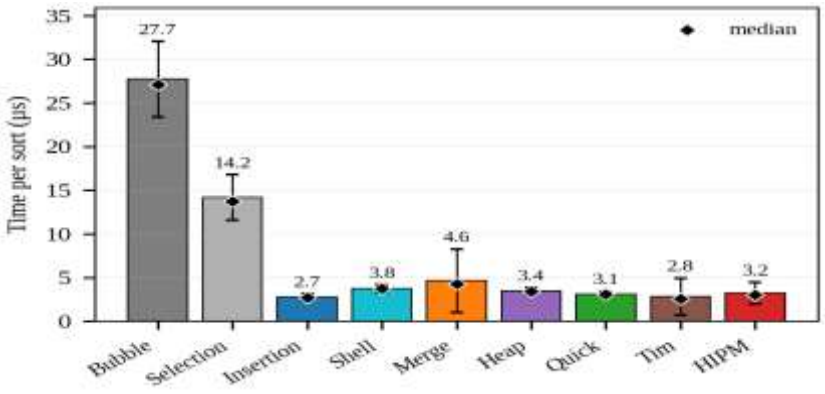


Fig. 3. Time chart for n = 100: mean time per sort (bars), ±1 standard deviation (whiskers) and median (diamonds).

Table III and Fig. 3 show three groups. Bubble sort (27.1 µs) and selection sort (13.7 µs) are 9× and 4.5× slower than HIPM Sort by median. Insertion sort (2.70 µs) and Tim sort (2.57 µs) are the fastest, about 11% and 15% ahead of HIPM Sort (3.03 µs). HIPM Sort is level with quick sort (3.05 µs) and ahead of heap (3.41 µs), Shell (3.71 µs) and merge sort (4.23 µs). The large standard deviations of merge, Tim and HIPM Sort come from single outliers (Table III, Max), not from systematic variation.

B. Scaling with Array Size

Table IV: Median Time per Sort in Microseconds versus Array Size

Algo.	50	100	200	500	1000	2000	5000	α
Bubble	7.4	27.1	93.8	518	1997	7846	48697	1.97
Selection	3.5	13.8	53.8	338	1369	5530	34923	2.01
Insertion	1.1	2.7	7.5	32.3	114	437	2628	1.91
Shell	1.5	3.8	8.7	26.6	60.4	138	402	1.18
Merge	1.8	4.2	9.4	27.1	59.8	130	362	1.13
Heap	1.5	3.4	6.7	18.7	39.9	85.7	236	1.10
Quick	1.4	3.1	6.8	19.7	43.6	96.2	269	1.14
Tim	1.0	2.6	6.2	19.1	43.7	98.0	284	1.17
HIPM	**1.4**	**3.0**	**6.8**	**21.9**	**56.1**	**169**	**819**	**1.58**

α: fitted log–log slope for n = 500–5000.

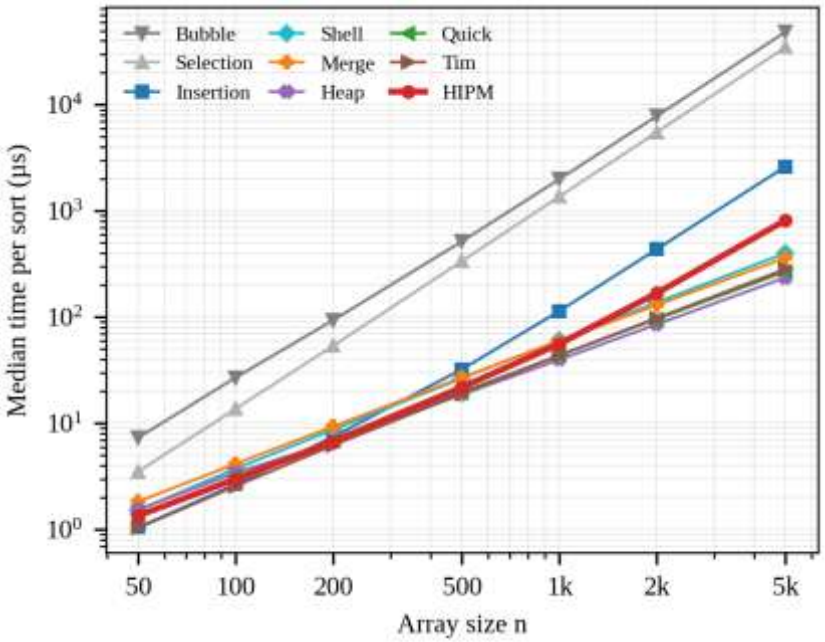


Fig. 4. Median time per sort versus array size (log–log). HIPM Sort (thick line) lies between the quadratic and the O(n log n) groups.

HIPM Sort overtakes insertion sort between n = 100 and 200 (6.8 versus 7.5 µs at n = 200) and beats bubble and selection sort at every size. Against the O(n log n) methods it is competitive up to n ≈ 500 (21.9 µs versus 18.7–27.1 µs), is slightly

ahead of merge and Shell sort at n = 1000 (56.1 µs versus 59.8 and 60.4 µs), but falls behind as the quadratic term takes over: at n = 5000 it needs 819 µs against 236 µs (heap), 269 µs (quick) and 363 µs (merge). Its fitted exponent ($\alpha = 1.58$) lies between the O(n log n) group (1.10–1.18) and the quadratic group (1.91–2.01) and is expected to rise toward 2 for larger n.

C. Effect of the Input Distribution

Table V: Median Time in Microseconds for Different Inputs (n = 1000)

Algorithm	Random	Sorted	Reversed	Nearly sorted	Few unique
Bubble	1997	1.1	3714	662	1802
Selection	1367	1324	1328	1329	1342
Insertion	114	1.1	216	17.9	105
Shell	60.1	5.8	10.3	43.0	35.0
Merge	59.5	14.3	14.8	22.3	43.1
Heap	40.1	29.6	27.7	33.0	41.6
Quick	44.1	375	344	36.7	63.6
Tim	43.7	3.7	21.8	11.1	27.5
HIPM	**56.2**	**4.1**	**65.1**	**19.0**	**46.7**

Quick sort with a fixed last-element pivot degrades to quadratic time on sorted and reversed data (375 and 344 µs versus 44 µs on random data), the weakness listed in Table I; selection sort is insensitive to the input (about 1330 µs); and bubble and insertion sort are excellent on sorted input (1.1 µs) but slow on reversed data (3714 and 216 µs). HIPM Sort is adaptive: it takes 4.1 µs on sorted and 19.0 µs on nearly sorted input, and its worst observed time (65.1 µs, reversed) is only 16% above its random-input time (56.2 µs). Tim sort is the best on nearly sorted data (11.1 µs), and Shell sort is best on reversed data (10.3 µs).

D. Ablation Study

Table VI: Ablation of HIPM Sort: Insertion Shifts and Median Time

Variant	Shifts n=100	µs n=100	Shifts n=1000	µs n=1000
Insertion sort	2,504	2.68	248,847	114.1
HIPM (full)	**642**	**2.99**	**70,170**	**55.9**
HIPM, no heap	1,024	2.66	104,599	63.6
HIPM, no pivot	867	2.88	90,605	62.8
Split + insert + merge	1,226	2.61	124,619	70.5

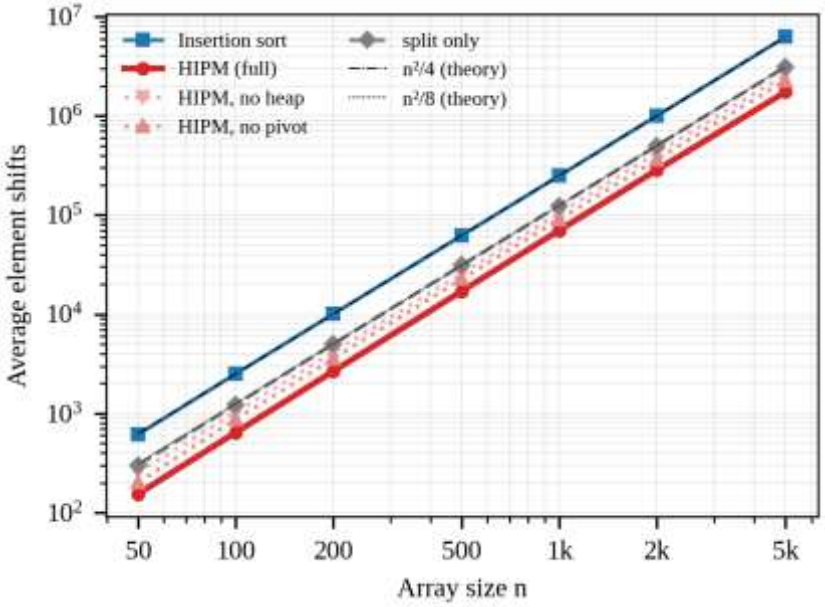


Fig. 5. Average number of element shifts of insertion sort and of HIPM variants versus n (log–log), with the theoretical $n^2/4$ and $n^2/8$ curves.

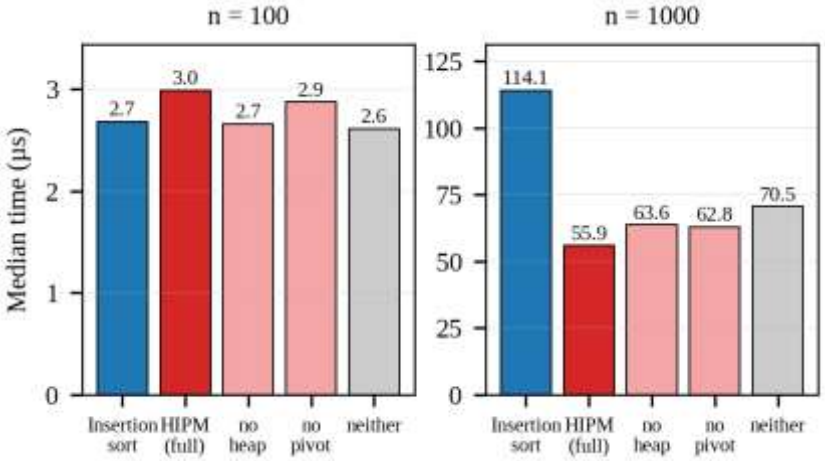


Fig. 6. Median time of HIPM Sort variants at n = 100 (left) and n = 1000 (right).

Splitting the array in two halves alone removes 51% of the shifts of insertion sort (2504 → 1226 at n = 100), close to the predicted $n^2/8 \approx 1250$. Heap construction alone removes a further 29% and the partition pass alone 17% of the split-only count, and together they leave only 642 shifts, 74% fewer than insertion sort (72% at n = 1000). Over n = 50–5000 the shifts of insertion sort follow $0.25\ n^2$ while those of HIPM Sort stay at $0.06–0.07\ n^2$ (Fig. 5). Whether this pays off in time depends on n: at n = 100 (Fig. 6, left) the extra O(n) passes cost more than they save (2.99 µs versus 2.61 µs for the plain split), whereas at n = 1000 (right) the full algorithm is 21% faster than the plain split (55.9 versus 70.5 µs) and 2.0× faster than insertion sort (114.1 µs).

E. Discussion and Threats to Validity

HIPM Sort is the fastest of the quadratic-class methods for $n \gtrsim 200$ and stays within about 10–20% of the best algorithm up to a few hundred elements, but it does not change the asymptotic class: for large arrays merge, heap, quick or Tim sort should be used. Results come from one machine and one compiler, use integer keys only, and rely on a simplified Tim sort, so absolute timings and the exact crossover points will differ on other systems; the ordering of the groups and the shift counts are the robust findings.

Table VII: Suggested Choice by Situation (from Tables I–VI)

Situation	Suitable choice	Reason
$n \lesssim 100$ or nearly sorted data	Insertion, Tim	Lowest overhead; adaptive (Tables III, V)
$n \approx 200$–1000, random data	HIPM, Tim, quick	Within about 40% of the best; no recursion (Table IV)
Large n, best average speed	Quick (introsort [6])	Lowest growth exponent; needs fallback for bad pivots (Tables IV, V)
Large n, worst-case guarantee, O(1) space	Heap	No bad inputs; not stable (Table II)
Large n, stability required	Merge, Tim	Stable and O(n log n) (Table II)

VII. Conclusion

This paper proposed HIPM Sort, a non-recursive hybrid algorithm that uses two insertion sorts with heap construction and a single pivot partition and then merges the halves. It is correct by construction, O(n) in the best case, $O(n^2)$ in the average and worst case, and removes about 74% of the element shifts of insertion sort. In the core benchmark (100 arrays of 100 integers) it is on par with quick sort and clearly ahead of bubble, selection, insertion, Shell and merge sort, while Tim sort are slightly faster; for n between 200 and 1000 it is up to twice as fast as insertion sort, and beyond a few thousand elements the O(n log n) algorithms win. Future work involves making the recursive procedure to handle large datasets and time series datasets [26] with O(n log n) complexity, potentially outperforming insertion sort when combined with merge-sort like in Shell sort and Tim sort.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[3] R. Sedgewick and K. Wayne, Algorithms, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[4] R. Sedgewick, “Implementing Quicksort programs,” Commun. ACM, vol. 21, no. 10, pp. 847–857, Oct. 1978.

[5] J. L. Bentley and M. D. McIlroy, “Engineering a sort function,” Softw., Pract. Exper., vol. 23, no. 11, pp. 1249–1265, Nov. 1993.

[6] D. R. Musser, “Introspective sorting and selection algorithms,” Softw., Pract. Exper., vol. 27, no. 8, pp. 983–993, Aug. 1997.

[7] T. Peters, “Timsort description (listsort.txt),” Python Software Foundation, CPython repository, 2002. [Online]. Available: https://github.com/python/cpython/blob/main/Objects/listsort.txt

[8] S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle, “OpenJDK’s java.utils.Collection.sort() is broken: The good, the bad and the worst case,” in Computer Aided Verification (CAV), ser. LNCS, vol. 9206. Cham, Switzerland: Springer, 2015, pp. 273–289.

[9] N. Auger, V. Jugé, C. Nicaud, and C. Pivoteau, “On the worst-case complexity of TimSort,” in Proc. 26th Annu. Eur. Symp. Algorithms (ESA), ser. LIPIcs, vol. 112, 2018, pp. 4:1–4:13.

[10] S. Edelkamp and A. Weiß, “BlockQuicksort: Avoiding branch mispredictions in Quicksort,” in Proc. 24th Annu. Eur. Symp. Algorithms (ESA), ser. LIPIcs, vol. 57, 2016, pp. 38:1–38:16.

[11] A. LaMarca and R. E. Ladner, “The influence of caches on the performance of sorting,” J. Algorithms, vol. 31, no. 1, pp. 66–104, Apr. 1999.

[12] V. Estivill-Castro and D. Wood, “A survey of adaptive sorting algorithms,” ACM Comput. Surv., vol. 24, no. 4, pp. 441–476, Dec. 1992.

[13] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro, “Insertion sort is O(n log n),” Theory Comput. Syst., vol. 39, no. 3, pp. 391–397, Jun. 2006.
[14] J. Katajainen, T. Pasanen, and J. Teuhola, “Practical in-place mergesort,” Nordic J. Comput., vol. 3, no. 1, pp. 27–40, 1996.
[15] O. Astrachan, “Bubble sort: An archaeological algorithmic analysis,” ACM SIGCSE Bull., vol. 35, no. 1, pp. 1–5, 2003.
[16] D. L. Shell, “A high-speed sorting procedure,” Commun. ACM, vol. 2, no. 7, pp. 30–32, Jul. 1959.
[17] A. A. Papernov and G. V. Stasevich, “A method of information sorting in computer memories,” Probl. Inf. Transm., vol. 1, no. 3, pp. 63–75, 1965.
[18] V. R. Pratt, Shellsort and Sorting Networks. Stanford University, 1972.
[19] R. Sedgewick, “A new upper bound for Shellsort,” J. Algorithms, vol. 7, no. 2, pp. 159–173, Jun. 1986.
[20] M. Ciura, “Best increments for the average case of Shellsort,” in Fundamentals of Computation Theory (FCT), ser. LNCS, vol. 2138. Berlin, Germany: Springer, 2001, pp. 106–117.
[21] J. W. J. Williams, “Algorithm 232: Heapsort,” Commun. ACM, vol. 7, no. 6, pp. 347–348, Jun. 1964.
[22] R. W. Floyd, “Algorithm 245: Treesort 3,” Commun. ACM, vol. 7, no. 12, p. 701, Dec. 1964.
[23] C. A. R. Hoare, “Algorithm 64: Quicksort,” Commun. ACM, vol. 4, no. 7, p. 321, Jul. 1961.
[24] C. A. R. Hoare, “Quicksort,” Comput. J., vol. 5, no. 1, pp. 10–16, 1962.
[25] R. Sedgewick and P. Flajolet, An Introduction to the Analysis of Algorithms, 2nd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2013.
[26] C. S. Ram, M. Raj, and R. K. Chaturvedi, “Boosting time-series forecasting accuracy with SARIMAX seasonal interval automation,” Procedia Computer Science, vol. 260, pp. 814-821, 2025.