Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

When Does a Theoretically Faster Algorithm Actually Win? An empirical crossover atlas for four classic algorithm pairs

dhanush paturu

1nt23cs060.dhanush@nmit.ac.in

Nitte University https://orcid.org/0009-0004-4918-6181

Research Article

Keywords: algorithm engineering, experimental algorithmics, asymptotic analysis, crossover point, radix sort, Strassen's algorithm, Karatsuba multiplication, Fibonacci heap

Posted Date: November 2nd, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11566676/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

When Does a Theoretically Faster Algorithm Actually Win? An empirical crossover atlas for four classic algorithm pairs

Dhanush Paturu*

Nitte Meenakshi Institute of Technology, Bengaluru, India

ORCID: https://orcid.org/0009-0004-4918-6181

*Corresponding author. Email: 1nt23cs060.dhanush@nmit.ac.in

Abstract

Asymptotic analysis promises that an algorithm with a better growth rate will eventually outperform a simpler one, but says nothing about *when*. For four classic pairs—radix sort versus introsort, Strassen versus naive matrix multiplication, Karatsuba versus schoolbook integer multiplication, and Dijkstra's algorithm with a Fibonacci heap versus a binary heap—we measure the input size at which the asymptotically faster algorithm first wins in wall-clock time on a current commodity CPU, and how that crossover moves with input structure (key width, sortedness, graph density) and with the tuning parameter every recursive algorithm has (its base-case cutoff). Both sides of each pair are plain, single-threaded C++ compiled with the same flags, and the recursive algorithms bottom out in exactly the kernel they are compared against. Crossovers are estimated on geometric size grids with bootstrap confidence intervals over interleaved repetitions. On an Intel Core i7-14650HX, radix sort overtakes introsort at about 1 500 keys regardless of key width, but never on nearly-sorted or reverse-sorted input, where introsort stays $3.7\times$ and $8\times$ ahead at 2^{26} keys; Strassen overtakes the naive triple loop from $n = 32$ on dimensions rich in factors of two but only from $n \approx 300$ on arbitrary dimensions, and the loop order of the *naive* baseline is worth more than Strassen's exponent; Karatsuba overtakes schoolbook multiplication at 34 limbs, within a factor of 1.3 of the threshold a tuned library uses on the same core family; and the Fibonacci heap never beats a binary heap on any sparse graph, reaching parity only on complete graphs at the edge of memory while still losing to the lazy-deletion binary heap that practitioners actually write. Crossovers are reported with bootstrap intervals and classified as sharp, gradual or marginal, and we argue that a wide interval is itself the practical answer: a range over which the choice does not matter. The result is a practical reference: a map of where the constants hidden by O-notation stop mattering, with the code and data to redraw it on any machine.

Keywords: algorithm engineering; experimental algorithmics; asymptotic analysis; crossover point; radix sort; Strassen's algorithm; Karatsuba multiplication; Fibonacci heap

1 Introduction

The standard argument for preferring an algorithm with a better asymptotic bound is that, for large enough inputs, it must win. The argument is correct and incomplete: it is silent about the size at which "large enough" begins, and in practice that size is set by constant factors that the analysis discards—the number of passes over memory, the size of the working set relative to the cache hierarchy, the predictability of branches, and the overhead of recursion and bookkeeping. Experienced programmers carry folk knowledge about these crossovers ("radix sort is only worth it for big arrays", "nobody uses Fibonacci heaps"), but the folk knowledge is rarely written down with numbers, and when it is, the numbers are a decade old and from a different cache hierarchy.

This paper measures the crossovers directly. For each of four algorithm pairs we sweep the input size over two and a half to seven orders of magnitude, vary the structure of the input and the tuning parameter of the faster algorithm, and record the first size at which the median wall-clock time of the theoretically faster algorithm drops below that of the simpler one. We call the resulting table and plots a *crossover atlas*.

Contributions.

1. A controlled, reproducible measurement of crossover points for radix sort vs. introsort, Strassen vs. naive matrix multiplication, Karatsuba vs. schoolbook multiplication and Fibonacci-heap vs. binary-heap Dijkstra, with bootstrap confidence intervals (Section 4).
2. An analysis of *what moves* each crossover: key width and sortedness for sorting, the base-case cutoff for the two recursive algorithms, and graph density and the lazy-deletion idiom for shortest paths (Section 4).
3. An account of the mechanism behind each crossover—passes over memory, the access pattern of the base case, pointer chasing against arrays—argued from working-set sizes and logged operation counts (Section 5).
4. A benchmarking harness that handles the practical difficulties (sub-microsecond inputs, cubic baselines that would otherwise dominate the sweep, drift between

repetitions) and that re-draws the atlas on any Linux machine with one command (Section 3, repository in Section 8).

2 Background and related work

2.1 Experimental algorithmics

The case for measuring algorithms rather than only analysing them was made systematically by Johnson (Johnson 2002), whose guide for theoreticians lists the pitfalls this paper tries to avoid (uninteresting instances, unfair baselines, unreported machines), and by McGeoch (McGeoch 2012), whose book is the standard reference on experimental design for algorithms. Sanders (Sanders 2009) frames the activity as *algorithm engineering*: a cycle of design, analysis, implementation and experiment in which the constant factors are first-class objects. The measurement side has its own literature of cautionary results. Mytkowicz et al. (Mytkowicz et al. 2009) showed that link order and environment-variable size alone can change measured runtimes by tens of percent, more than many published effects; Curtsinger and Berger (Curtsinger and Berger 2013) proposed randomising memory layout so that such effects average out; and Hoefler and Belli (Hoefler and Belli 2015) surveyed reporting practice and argued for repetition, interval estimates and full disclosure of the machine. LaMarca and Ladner (LaMarca and Ladner 1999) is the direct ancestor of this study for sorting: by simulating cache behaviour they showed that the instruction count of a sorting algorithm is a poor predictor of its time once the input leaves cache, and that memory-aware variants of heapsort and mergesort outrun the textbook versions.

2.2 The four pairs

Sorting.

Introsort (Musser 1997)—quicksort with a heapsort fallback and insertion sort for small ranges—is what `std::sort` runs in every major C++ standard library, which makes it the baseline a practitioner actually faces. Rahman and Raman (Rahman and Raman 2001) adapted LSD radix sort to the cache and TLB and showed it beating comparison sorts on large arrays of integers; Satish et al. (Satish et al. 2010) compared radix and merge sort across CPUs and GPUs and found radix sort faster for 32-bit keys but merge sort preferable for wide keys and on bandwidth-limited hardware; Polychroniou and Ross (Polychroniou and Ross 2014) studied the partitioning step that both families share at billion-key scale. None of these report

the crossover size against `std::sort` as an estimated quantity, and all use SIMD and multithreading that move it.

Matrix multiplication.
Strassen's 1969 algorithm (Strassen 1969) is the archetype of an asymptotic improvement that is practical only after engineering. Huss-Lederman et al. (Huss-Lederman et al. 1996) gave an influential careful implementation study, establishing that the recursion must stop at a tuned cutoff and that odd dimensions need peeling or padding; D'Alberto and Nicolau (D'Alberto and Nicolau 2007) tuned that cutoff adaptively to the memory hierarchy; Benson and Ballard (Benson and Ballard 2015) showed that the best fast algorithm depends on matrix shape; and Huang et al. (Huang et al. 2016), by fusing Strassen's additions into a BLIS kernel, brought the *switch point*—the dimension above which one level of Strassen pays off against a vendor-grade DGEMM—down from the roughly 2 000 of earlier implementations to as little as 500 for one of their implementations, on a single Ivy Bridge Xeon core. Those crossovers are against the strongest possible baseline. Ours is against the baseline a programmer writes without a BLAS, and is correspondingly much earlier.

Integer multiplication.
Karatsuba and Ofman's algorithm (Karatsuba and Ofman 1962) is the first sub-quadratic multiplication and the first step of the tower that ends, for now, at Harvey and van der Hoeven's $\mathcal{O}(n\log n)$ algorithm (Harvey and van der Hoeven 2021), which is correct and unimplementable. Bernstein (Bernstein 2001) gives the modern unified view. The GNU Multiple Precision library (Granlund and GMP development team 2023) switches from schoolbook to Karatsuba at a machine-tuned threshold (`MUL_TOOM22_THRESHOLD`), which its manual notes can be as low as 10 limbs; the parameter file GMP ships for Alder Lake sets it to 13 limbs (Granlund and GMP development team 2022). This gives an external reference point for our measured crossover.

Shortest paths.
Fredman and Tarjan (Fredman and Tarjan 1987) introduced the Fibonacci heap precisely to improve Dijkstra's algorithm from $\mathcal{O}(m\log n)$ to $\mathcal{O}(m + n\log n)$, and the data structure has been a textbook fixture ever since; Brodal, Lagogiannis and Tarjan (Brodal, Lagogiannis, and Tarjan 2012) later made the bounds worst-case. The empirical record is uniformly unkind to it. Cherkassky, Goldberg and Radzik (Cherkassky, Goldberg, and Radzik 1996) found that no single shortest-path code was best on all of their problem families; Chen et al. (Chen et al.

2007) dropped the Fibonacci heap from their comparison after preliminary runs and found that Dijkstra variants *without* decrease-key were at least $1.4\times$ faster than those with it on random graphs of average degree 8 (and faster on every road network they tried); and Larkin, Sen and Tarjan (Larkin, Sen, and Tarjan 2014), on the full USA road network, measured the Fibonacci heap at $3.15\times$ the time of an implicit 4-ary heap, with one decrease-key for every fourteen inserts, and found the L1 cache-miss rate to be the counter most strongly correlated with wall-clock time. Our results agree with all of these and add the one regime, complete graphs, in which the asymptotic argument finally holds.

2.3 Hidden constants and galactic algorithms

Lipton (Lipton 2010) introduced the term *galactic algorithm*, a name he credits to Ken Regan, for one whose asymptotic advantage appears only on inputs larger than any that will be run, with fast matrix multiplication as the canonical example: the exponent has been pushed to $\omega < 2.372$ (Alman et al. 2025; the preceding bound, $\omega < 2.3729$, is due to Alman and Vassilevska Williams 2024), yet nothing past Strassen is used in practice. This paper sits at the opposite end of that spectrum. Its four pairs are algorithms that are actually used, and the question is not whether their crossovers exist but where, with what precision, and what moves them. The hardware background that answers "why"—cache levels, memory-level parallelism, branch prediction—is laid out by Drepper (Drepper 2007) and Fog (Fog 2026); the Linux `perf_event_open` interface (Linux man-pages project 2026) is how the harness reads the counters where it can.

3 Method

3.1 Algorithm pairs and what is held fixed

Each pair consists of a *faster-in-theory* algorithm F and a *simpler* algorithm S (Table 1). Three rules keep the comparison about the algorithmic idea rather than about implementation effort.

1. **Same language, compiler and flags.** C++17, GCC, `-O2`, no `-march=native` (the x86-64 baseline ISA), no hand-written SIMD, no threads. (The harness can rebuild with `-O3 -march=native` for a secondary experiment; that experiment is not part of this paper.)

2. **Same base case.** The recursive algorithms (Strassen, Karatsuba) bottom out in exactly the kernel they are compared against, so the measured difference is attributable to the recursive structure, its extra additions and its memory traffic.

3. **Competent, textbook implementations on both sides.** Radix sort computes all digit histograms in one pass and skips passes in which every key shares a digit; Karatsuba uses a caller-provided scratch buffer; the Fibonacci heap pre-allocates one node per vertex. The simpler algorithms are what a careful programmer writes: `std::sort`, the i-k-j loop order, 64-bit multiply-accumulate, an indexed binary heap with decrease-key—and, as a second baseline, the lazy `std::priority_queue` idiom that is far more common in practice.

Table 1. The four algorithm pairs and the variables swept.

Pair	F	S	Swept
Sorting	LSD radix sort, $\mathcal{O}(n \cdot 64/d)$ for d-bit digits	introsort (`std::sort`), $\mathcal{O}(n\log n)$	$n \in [16, 2^{26}]$; key width 8/16/32/64 bits; nearly-sorted, reverse, few-unique, organ-pipe; $d \in \{8,11,16\}$
Matrix multiply	Strassen, $\mathcal{O}(n^{2.807})$	naive triple loop, $\mathcal{O}(n^3)$	$n \in [16,4096]$ incl. non-powers of two; cutoff $\in \{8, \ldots, 256\}$; i-k-j vs i-j-k baseline
Integer multiply	Karatsuba, $\mathcal{O}(n^{1.585})$	schoolbook, $\mathcal{O}(n^2)$	$n \in [8, 2^{20}]$ 32-bit limbs; cutoff $\in$ $\{8, \ldots, 128\}$; random and all-ones limbs
Shortest paths	Dijkstra + Fibonacci heap, $\mathcal{O}(m + n\log n)$	Dijkstra + binary heap, $\mathcal{O}((n + m)\log n)$	$n \in [10^2, 8.4 \times 10^6]$; average degree $\in \{2, \ldots, 256\}$; random / 2-D grid / complete graphs; lazy baseline

3.2 Inputs

Inputs are generated deterministically from a 64-bit seed combined with n (xoshiro256** seeded through splitmix64), so every cell of the experiment is reproducible from the command line that produced it. Sizes form a geometric grid (factor 2 for the main sweeps, 1.5 for integer multiplication, and a hand-picked list for matrices that includes sizes such as 24, 48 and 96 that exercise Strassen's padding path). A *cell* is a (pair, input family, parameter, n) tuple. Cells that make no sense are skipped by rule (a complete graph on 10^7 vertices); an algorithm whose single run exceeds a wall-clock budget is dropped for larger n in that series, so that cubic baselines do not dominate the running time of the experiment.

3.3 Measurement

Timer.

`std::chrono::steady_clock` (`CLOCK_MONOTONIC` through the vDSO, about 20 ns overhead). For inputs whose single run exceeds 2 ms the input is copied outside the timed

region and only the algorithm is timed. Shorter runs are batched: the harness runs K iterations of (copy input; run algorithm) and reports the per-iteration mean, with K chosen from a warm calibration run; because in-place algorithms need a fresh copy per iteration, the copy is inside the timed region, and its cost is measured separately with the same K on the same pre-allocated buffer and subtracted in the analysis (never below 5% of the raw time).

Repetitions and interleaving.
Each cell is measured $r = 5$ times after $w = 2$ untimed warm-up runs. Within the repetition loop the algorithms alternate in a rotating order, so slow drift in machine state (thermal throttling, background activity, frequency changes) is shared equally by F and S. The per-cell statistic is the median; the interquartile range is retained and plotted.

Hardware counters.
When `perf_event_open` is permitted, each measurement also records cycles, retired instructions, L1d read misses, last-level-cache misses and branch mispredictions for the user-space portion of the run, scaled for multiplexing. These are explanatory, not primary: the crossover is defined on wall-clock time. Where counters are unavailable (virtual machines, WSL2) the study is timing-only and says so.

Correctness.
Every benchmark binary cross-checks the outputs of F and S on the last repetition of each cell and aborts on mismatch. Separate randomized test suites compare each implementation with an independent reference (`std::sort`; the i-j-k product; schoolbook plus the modular identity $(ab) \bmod p = \big((a \bmod p)(b \bmod p)\big) \bmod p$ with $p = 2^{61} - 1$; Bellman–Ford), and both heaps are exercised directly against a scan-for-minimum oracle under random push, decrease-key and pop sequences.

3.4 Runs and a measurement correction

The reported dataset is the union of three runs made in one session on the same machine, with the same binary flags: `full` (the sweeps of Table 1, 5.3 hours), `refine` (factor-1.19 grids over the neighbourhood of each crossing, 20 minutes) and `patch` (70 minutes). The patch exists because the first version of the harness measured the input-copy baseline into a freshly allocated buffer, so that for large inputs the baseline included the operating system's page-fault cost while the copy inside the timed loop, reusing the buffer, did not; the subtraction then removed too much. The effect is below 5% wherever the copy is small relative to the

run, which is almost everywhere, but for complete graphs (470 MB of adjacency at 7 700 vertices against a 100 ms run) and for the largest arrays sorted by the fastest configurations it was large enough to invalidate the cell. We fixed the harness as described above, re-measured every series in which the copy had exceeded 5% of the raw time (dense graphs at all sizes, arrays of 2^{21} keys and above, sparse graphs of 2^{20} vertices and above, and a handful of small cells), and let the re-measurement *replace* the original cells over the range of n it covers rather than pool with them. Both the original and the corrected data are in the repository; the analysis script's `-supersede` flag applies the replacement.

3.5 Machine

All reported measurements were made on a laptop with an Intel Core i7-14650HX (Raptor Lake, 8 performance cores + 8 efficiency cores, 24 threads; 48 KB L1d and 2 MB L2 per performance core, 30 MB shared L3) running Ubuntu under WSL2 (Linux 6.18, Microsoft hypervisor) with 7.6 GB of memory visible to the guest, GCC 15.2.0 with `-O2`, CMake 4.2. The laptop was on mains power in the manufacturer's *Performance* mode, the Windows power plan was set to *High performance* with maximum processor state 99% (which disables Turbo Boost on Intel parts), the WSL virtual machine was restricted to the performance cores through its process affinity, and the benchmark thread was pinned to one virtual CPU. The hypervisor hides hardware performance counters, so the reported run is timing-only; the harness collects counters automatically on bare-metal Linux, where it also sets the performance governor, disables turbo and pins the frequency (`scripts/setup_machine.sh`). We report the machine exactly and treat absolute times as specific to it; the crossover *ordering* and *order of magnitude* are the transferable results.

3.6 Crossover estimation

For a series (pair, family, parameter) let $\rho(n) = \tilde{t}_F(n)/\tilde{t}_S(n)$ be the ratio of median runtimes on the size grid. The crossover n^* is the *sustained* crossing: the grid interval after which $\rho < 1$ at every larger measured size, interpolated linearly in $(\log n, \log \rho)$. A transient dip below 1 followed by a return above it is not counted; it remains visible in the ratio plots and in the reported minimum ratio. Series with $\rho < 1$ already at the smallest n are reported as *always faster* (from $n_{\min}$) and series with $\rho \geq 1$ throughout as *never faster* (up to $n_{\max}$), together with $\rho(n_{\max})$ so the reader can see how far from crossing they are. Uncertainty comes from a non-parametric bootstrap: the r repetitions of every cell are resampled with replacement (1000 replicates), medians and n^* are recomputed, and the shortest interval containing 95% of

the replicates (in $\log n$) is reported. The shortest rather than the central interval is used because the sustained-crossing statistic can be bimodal: a resample in which one noisy mid-range cell pops above $\rho = 1$ relocates the crossing to the far end of the sweep, and a percentile interval would stretch over that tail even when nearly all of the mass sits tightly around the true crossing.

Two caveats govern how the intervals should be read. First, n^* is interpolated between grid points, so an interval cannot resolve a crossing more finely than the grid allows; the main sweeps use a factor-2 grid and the `refine` profile re-sweeps the neighbourhood of each crossing on a factor-1.19 grid for that purpose. Second, and more important, the width of an interval is itself a result. We classify each crossing as *sharp* when its interval is narrower than one grid step or than a factor of 1.5, whichever is wider (a $\pm 20\%$ interval is sharp by any practical standard, even on a fine grid), as *gradual* otherwise, and as *marginal* when the faster algorithm is never more than 10% ahead by the largest size measured, whatever the interval. A gradual crossing means that ρ approaches 1 slowly, i.e. the two algorithms are within measurement noise of each other across the whole interval; the honest statement is then not "the fast algorithm wins from n^*" but "over this range it makes no measurable difference, and beyond it the fast algorithm wins by the ratio we report at $n_{\max}$". Gradual crossings are reported as ranges, never as points. Empirical scaling exponents are fitted as $\log t = b \log n + c$ over the largest decade of n for each algorithm, to show how far the measured growth is from the asymptotic bound in the measured range.

3.7 Use of generative AI tools

Claude Opus 5 (Anthropic) was used to assist with the literature review and with grammar and wording revisions of the manuscript. All measurements, analyses and conclusions are the author's own; the author reviewed every AI-assisted passage, checked each cited source against the original publication, and takes full responsibility for the content of this paper.

4 Results

All numbers in this section come from the `full` profile (factor-2 size grids, 5 timed repetitions after 2 warm-ups, 2 411 cells, 5.3 hours) on the machine of Section 3, supplemented by the `refine` profile (factor-1.19 grids around the crossings) and the `patch` profile described in Section 3.4. Table 2 gives every crossing with its interval and class.

4.1 Sorting: radix vs. introsort

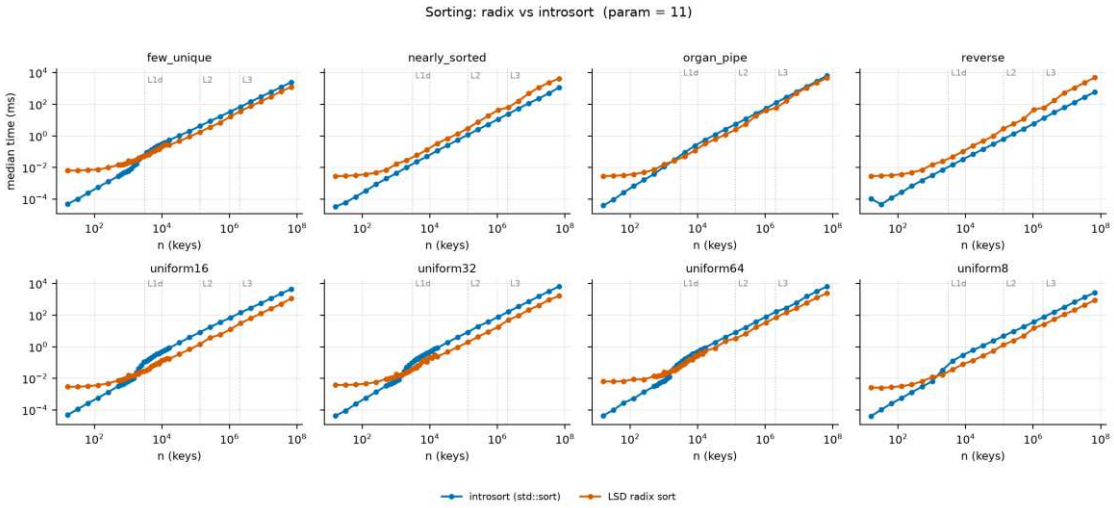


Figure 1. Median runtime of LSD radix sort (11-bit digits) and introsort by input family. Dotted lines mark the n at which the working set ($16n$ bytes) equals L1d, L2 and L3. Radix sort sits on a flat floor below $n \approx 10^3$, where the histogram pass dominates.

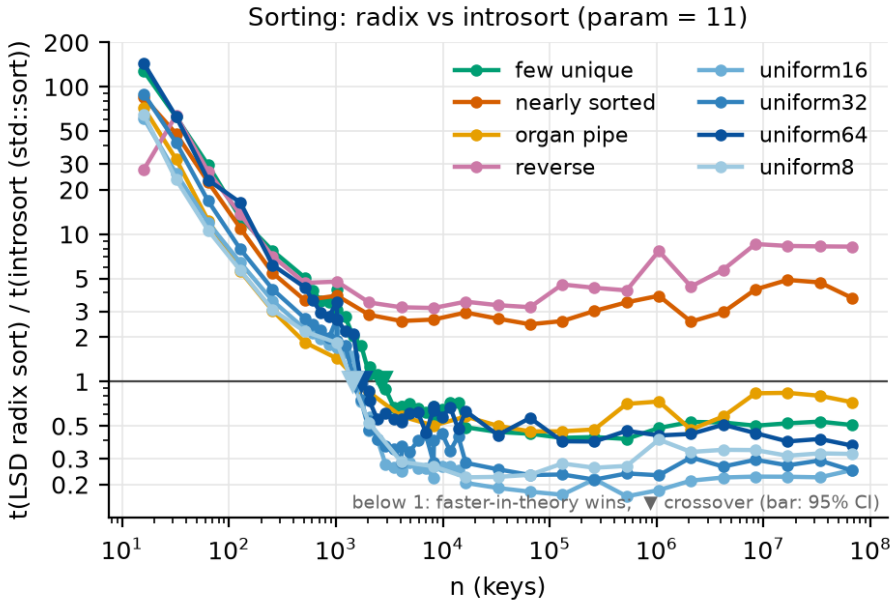


Figure 2. Runtime ratio radix/introsort by input family (11-bit digits). Uniform keys of four widths cross between 1.4 k and 1.7 k; nearly-sorted and reverse-sorted input never cross.

On uniformly random keys radix sort wins from about 1 500 elements and the crossing is sharp (Figures 1 and 2): 1.43 k keys for 8-bit values, 1.60 k for 16-bit, 1.51 k for 32-bit and 1.71 k for 64-bit, each with an interval of $\pm 10\%$ or less. The near-independence from key width is itself a finding: with 11-bit digits a 64-bit key needs six passes and an 8-bit key one, yet the crossing barely moves, because below $\sim 10^3$ keys both algorithms are dominated by fixed costs—the histogram and prefix-sum setup on one side, insertion-sort leaves on the

other—and above it both scale as n to within a few percent over the whole range (fitted exponents 1.00–1.06 for radix, 1.02–1.14 for introsort). Above the crossing the advantage settles between $2.7\times$ (64-bit keys: 2.44 s against 6.63 s at 2^{26}) and $4\times$ (16-bit keys: 1.14 s against 4.55 s).

Digit width is the practitioner's knob and it is not free. Eleven-bit digits are best at every key width; eight-bit digits cost one or two extra passes and move the 64-bit crossing only to 1.67 k, but sixteen-bit digits are a mistake for wide keys: the 2^{16}-counter histogram per pass is 512 KB, larger than L1d, and at 64-bit keys the crossing moves to 7 k and the asymptotic advantage shrinks to $1.4\times$ (the gradual interval reaching 8.6 M in Appendix A reflects the ratio hovering between 0.5 and 0.75 across the whole range).

The structured families overturn the verdict. With only 100 distinct values introsort's partitioning handles duplicates well and the crossing moves to 2.7 k with an eventual advantage of only $2\times$. On nearly-sorted input radix sort *never* wins: introsort is $3.7\times$ faster at 2^{26} keys (1.15 s vs 4.26 s). On reverse-sorted input it is $8\times$ faster (0.60 s vs 4.92 s)—median-of-three pivot selection turns a reversed array into perfectly balanced partitions with perfectly predictable branches, and introsort runs $11\times$ faster on it than on random data of the same size. The organ-pipe input, ascending then descending, is the intermediate case: radix sort crosses at 1.6 k but only ever gets $1.4\times$ ahead. Radix sort performs the same scattered writes whether the input is sorted or not; an algorithm that is oblivious to structure cannot exploit it.

4.2 Matrix multiplication: Strassen vs. naive

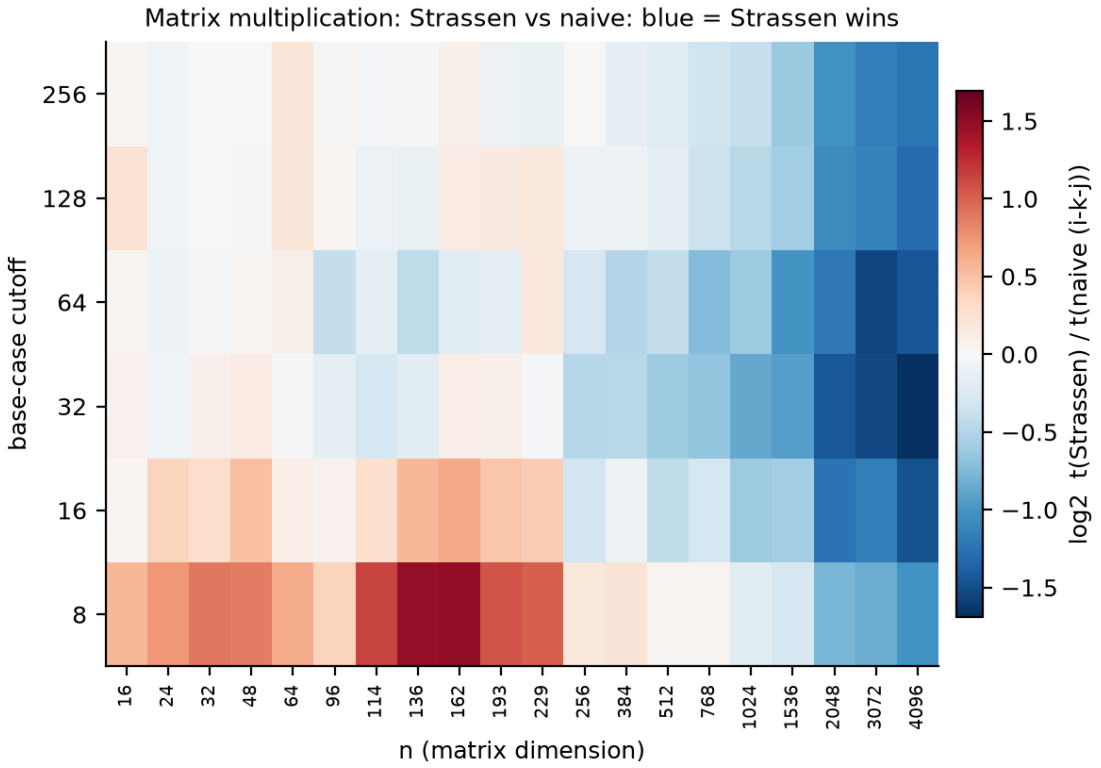


Figure 3. $\log_2$ of the runtime ratio Strassen/naive (i-k-j) over matrix size and base-case cutoff on the power-of-two-rich grid; blue means Strassen wins.

Strassen's algorithm is practical much earlier than folklore suggests, with one important qualification about matrix shape. On sizes with many factors of two ($n = 2^k$ and $3 \cdot 2^k$, the grid every benchmark uses) and a base-case cutoff of 32, it beats the i-k-j triple loop at every size from $n = 32$ up—0.83 × at $n = 64$, 0.71 × at 256, 0.39 × at 2048, 0.31 × at 4096 (13.4 s vs 43.3 s)—and the fitted exponent over the top decade is 2.78 against the theoretical 2.81, with the naive loop at 3.21.

The refine grid, which includes sizes such as 129, 154, 217 and 259, tells a different story. Our implementation handles an odd dimension at any level of the recursion by padding that level to even size, which is the textbook approach; an n with few factors of two is padded at level after level, and every padding copies both operands. On those sizes Strassen with cutoff 32 is up to 1.2 × *slower* than naive until $n \approx 260$, and the sustained crossing over all measured sizes is $n \approx 300$ (interval $[231,308]$), five times later than on the composite grid. Both numbers are right; which one applies depends on whether the caller controls the matrix dimensions. Production implementations pad or peel once at the top level, which would remove this effect, and the comparison shows how much it is worth.

The cutoff behaves as expected (Figure 3): 8 is one level too deep and costs a factor of 1.6 at $n = 4096$ (21.2 s vs 13.4 s), with the crossing pushed out to between 780 and 1 300 on uniform matrices (gradual) and to 811 on integer ones; 16 is 1.2 × slower than naive at $n = 32$; 32 and 64 are equivalent within noise at the top of the sweep; 128 and 256 forgo the advantage below 512 and are 30–35% slower than cutoff 32 at the top. Integer-valued matrices behave the same as uniform real ones to within noise, as they should for an algorithm whose work does not depend on the values.

The second baseline is the cautionary example. The textbook i-j-k loop order, which walks the second operand down a column with stride n, has a fitted exponent of 4.3 over $n \in [229,2048]$—it is not cubic in practice, for reasons we take up in Section 5—and at $n = 2048$ it takes 84 s where i-k-j takes 5.0 s and Strassen 1.9 s. Swapping two loop headers in the *simple* algorithm is worth 17 × at that size; Strassen's exponent, applied on top, is worth another 2.6 ×.

4.3 Integer multiplication: Karatsuba vs. schoolbook

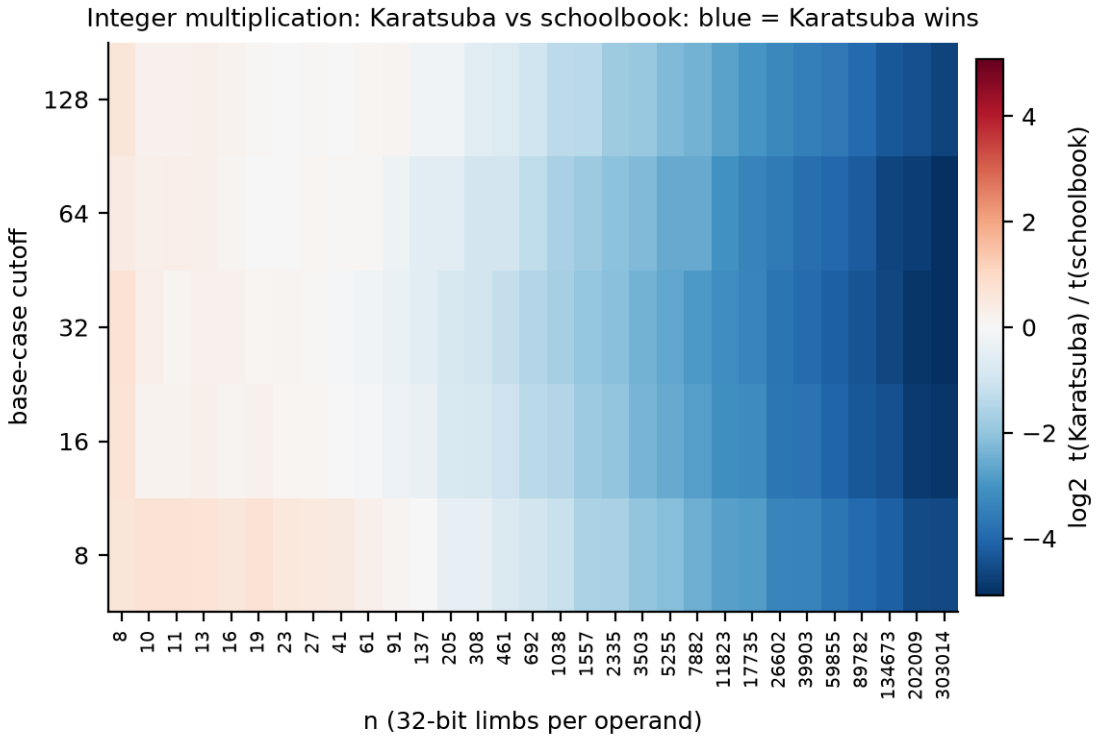


Figure 4. $\log_2$ of the runtime ratio Karatsuba/schoolbook over operand length (32-bit limbs) and cutoff.

Karatsuba is the cleanest case. With a cutoff of 32 limbs it overtakes schoolbook multiplication at 34 limbs (interval [32,42]; roughly 1 100 bits), essentially as soon as it does

anything at all, and the measured exponents over the top decade are 1.55–1.60 for Karatsuba against 1.97–2.01 for schoolbook, within 0.04 of $\log_2 3$ and 2. Schoolbook exceeded the 90-second budget at 4.5×10^5 limbs; at the last common size, 3×10^5 limbs (9.7 Mbit operands), Karatsuba is $18 \times$ faster (3.6 s vs 65 s). The crossing at 34 32-bit limbs is 17 64-bit limbs (about 1 090 bits). GMP's tuned parameter file for Alder Lake, the previous generation of the same core design, sets `MUL_TOOM22_THRESHOLD` to 13 64-bit limbs (830 bits) (Granlund and GMP development team 2022), so the two land within a factor of 1.3 of each other. The agreement should not be over-read: GMP measures its threshold against a hand-written assembly basecase and with an assembly Karatsuba, so both sides of its comparison differ from ours, and the near-coincidence says mainly that the crossover is insensitive to the quality of the implementation when both sides are implemented with equal care.

With all-ones limbs—maximal carry propagation—schoolbook is unchanged but Karatsuba is $1.8 \times$ faster than on random limbs, because the sub-products of the sum terms have predictable carries; the crossing is essentially unchanged (Karatsuba is ahead from the first size at which it differs from schoolbook). The cutoff sensitivity is asymmetric (Figure 4): cutoffs of 32, 64 and 128 are indistinguishable at large n, 16 makes the crossing gradual (the two are within noise from 34 to about 60 limbs), and 8 pushes it out to ≈ 700 limbs and costs a factor of 1.9 at every size above (42.8 s vs 22.7 s at 10^6 limbs). A cutoff four times too large costs nothing; four times too small costs $2 \times$.

4.4 Shortest paths: Fibonacci heap vs. binary heap

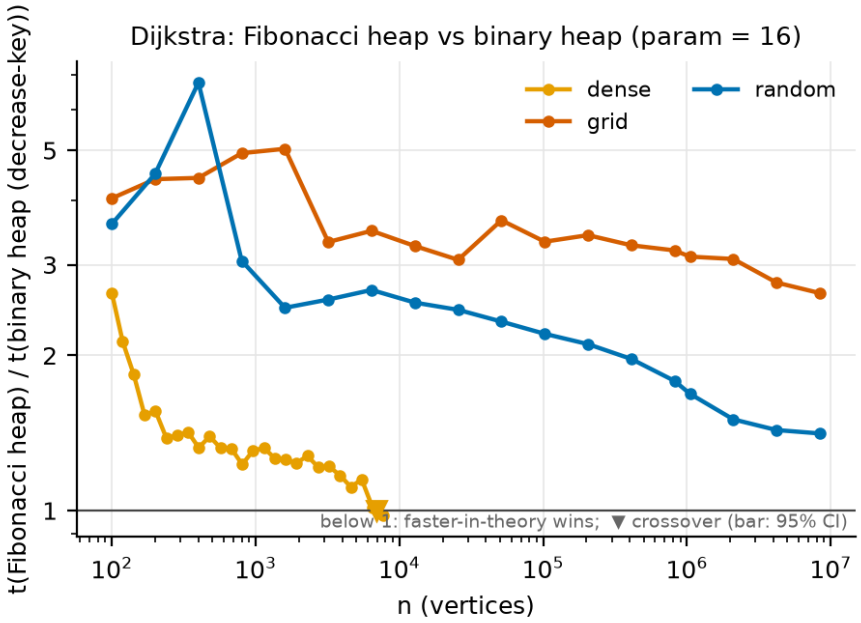


Figure 5. Runtime ratio Fibonacci-heap/binary-heap (decrease-key) Dijkstra by graph family (random graphs: average out-degree 16). Only the complete graph reaches the line, and only at the largest size that fits in memory.

The Fibonacci heap is the pair that does not cross. On random graphs of average degree 2, 4, 16, 64 and 256 it trails the indexed binary heap by 2.1 ×, 1.7 ×, 1.4 ×, 1.3 × and 1.3 × at the largest sizes measured (8.4×10^6 vertices at degrees 2–16, 134 M edges at degree 16; 4.2×10^6 at degree 64 and 1.0×10^6 at 256), on 2-D grids by 2.6 × (in line with the 3.15 × that Larkin, Sen and Tarjan measured against an implicit 4-ary heap on the USA road network (Larkin, Sen, and Tarjan 2014)), and the gap narrows only slowly with n (Figure 5). The reason is in the operation counts the harness logs: Dijkstra performs 0.05 decrease-keys per vertex on degree-2 graphs, 0.3 on degree-4 graphs and grids, and 1.4 at degree 16, so the $\mathcal{O}(1)$ amortised decrease-key that justifies the Fibonacci heap is exercised far less often than the $\mathcal{O}(\log n)$ delete-min that both heaps share; Section 5 argues that the Fibonacci heap's delete-min is the more expensive of the two.

On complete graphs ($m = n(n-1)$, 6 decrease-keys per vertex) the ratio falls steadily from 2.4 at 100 vertices to 1.0 at 6 500 and 0.98 at 7 700 vertices (60 M edges, the largest that fits in memory). This is the regime $m \gg n\log n$ in which $\mathcal{O}(m + n\log n)$ should beat $\mathcal{O}(m\log n)$, and the trend is as theory predicts—but what it reaches within the measurable range is parity, not a win, and the atlas reports it as such.

Even that parity is against the wrong baseline. The lazy variant, `std::priority_queue` with stale entries skipped on pop, does no decrease-keys at all and is the fastest of the three on every random graph and on the complete graph: 78 ms against 135 ms (Fibonacci) and 138 ms (indexed binary) at 7 700 dense vertices; 6.3 s against 15.0 s and 8.6 s at 8.4 M vertices of degree 4—the same "at least 1.4 ×" advantage for dropping decrease-key that Chen et al. report on random graphs of average degree 8 (Chen et al. 2007). Its heap holds duplicate entries and does more pops, but each pop touches a contiguous array and the extra work costs less than any form of decrease-key bookkeeping. Only on grids, where the frontier is small and duplicates accumulate, does the indexed binary heap edge it out (1.89 s vs 2.28 s). The Fibonacci heap never beats the lazy heap in any cell, trailing it by 1.7 × even on the complete graph.

4.5 Summary

Table 2. Sustained crossover points with 95% bootstrap intervals (shortest interval over 1000 resamples of the repetitions). Sharp crossings are given as a point; gradual ones (interval wider than a grid step or a factor 1.5) as the range over which the two algorithms are within noise of each other; "parity only" marks a crossing that is never more than 10% ahead by the largest size measured. $\rho(n_{\max})$ is the runtime ratio at the largest size measured. For matrices the crossing is reported on the power-of-two-rich grid and on all sizes separately. This table shows the default parameter of each pair (11-bit digits, cutoff 32); every parameter is in Appendix A.

Pair	Input	Param	Comparison	n^*	95% CI	Outcome	$\rho(n_{\max})$
sorting	few unique	11	radix vs introsort	2.7 k	[2.4 k, 2.8 k]	crosses	0.51
sorting	nearly sorted	11	radix vs introsort	–	–	never faster to 67.1 M	3.69
sorting	organ pipe	11	radix vs introsort	1.6 k	[1.5 k, 1.8 k]	crosses	0.72
sorting	reverse	11	radix vs introsort	–	–	never faster to 67.1 M	8.23
sorting	uniform16	11	radix vs introsort	1.6 k	[1.6 k, 1.6 k]	crosses	0.25
sorting	uniform32	11	radix vs introsort	1.5 k	[1.4 k, 1.6 k]	crosses	0.25
sorting	uniform64	11	radix vs introsort	1.7 k	[1.6 k, 2.1 k]	crosses	0.37
sorting	uniform8	11	radix vs introsort	1.4 k	[1.4 k, 1.5 k]	crosses	0.32
matmul	integer (2^k, 3·2^k)	32	Strassen vs naive i-k-j	32–100	[32, 100]	crosses (gradual)	0.31
matmul	uniform (2^k, 3·2^k)	32	Strassen vs naive i-k-j	≤ 32	–	faster at every n	0.31
matmul	integer (all n)	32	Strassen vs naive i-k-j	218	[202, 235]	crosses	0.31
matmul	uniform (all n)	32	Strassen vs naive i-k-j	297	[231, 308]	crosses	0.31
matmul	integer (2^k, 3·2^k)	32	Strassen vs naive i-j-k	32–89	[32, 89]	crosses (gradual)	0.03
matmul	uniform (2^k, 3·2^k)	32	Strassen vs naive i-j-k	≤ 32	–	faster at every n	0.02
matmul	integer (all n)	32	Strassen vs naive i-j-k	32–89	[32, 89]	crosses (gradual)	0.03
matmul	uniform (all n)	32	Strassen vs naive i-j-k	59	[48, 69]	crosses	0.02
bigint	ones	32	Karatsuba vs schoolbook	≤ 41	–	faster at every n	0.03
bigint	uniform	32	Karatsuba vs	34	[32, 42]	crosses	0.05

Pair	Input	Param	Comparison	n^*	95% CI	Outcome	$\rho(n_{\max})$
			schoolbook				
dijkstra	dense	-	Fib. heap vs binary (dec-key)	≈7.0 k	[6.5 k, 7.6 k]	parity only	0.98
dijkstra	grid	-	Fib. heap vs binary (dec-key)	–	–	never faster to 8.4 M	2.64
dijkstra	random	16	Fib. heap vs binary (dec-key)	–	–	never faster to 8.4 M	1.41
dijkstra	random	2	Fib. heap vs binary (dec-key)	–	–	never faster to 8.4 M	2.09
dijkstra	random	256	Fib. heap vs binary (dec-key)	–	–	never faster to 1.0 M	1.29
dijkstra	random	4	Fib. heap vs binary (dec-key)	–	–	never faster to 8.4 M	1.73
dijkstra	random	64	Fib. heap vs binary (dec-key)	–	–	never faster to 4.2 M	1.30
dijkstra	dense	-	Fib. heap vs binary (lazy)	–	–	never faster to 7.7 k	1.73
dijkstra	grid	-	Fib. heap vs binary (lazy)	–	–	never faster to 8.4 M	2.19
dijkstra	random	16	Fib. heap vs binary (lazy)	–	–	never faster to 8.4 M	1.52
dijkstra	random	2	Fib. heap vs binary (lazy)	–	–	never faster to 8.4 M	2.52
dijkstra	random	256	Fib. heap vs binary (lazy)	–	–	never faster to 1.0 M	1.35
dijkstra	random	4	Fib. heap vs binary (lazy)	–	–	never faster to 8.4 M	2.38
dijkstra	random	64	Fib. heap vs binary (lazy)	–	–	never faster to 4.2 M	1.22

Three of the four theoretically faster algorithms win, and early: at 1 500 keys, 32 rows, and 34 limbs—and for the well-tuned configurations these are sharp crossings with intervals of $\pm 10\%$. Each has conditions: radix sort needs the input not to be nearly or reverse sorted, Strassen needs the matrix dimension to be rich in factors of two or the implementation to pad only once, Karatsuba needs its cutoff not to be too small. The fourth reaches parity on complete graphs at the edge of memory and loses everywhere else, including to the lazy idiom that practitioners actually write.

5 Mechanisms

The reported run was made under WSL2, which hides the hardware performance counters, so what follows is not measured: it is the set of explanations we find most consistent with the timings, the working-set sizes and the operation counts the harness logs. Each is stated as an attribution and could be tested directly with the counter support the harness already has on bare-metal Linux (`figures/fig_counters_*` are produced automatically from such a run).

Sorting: passes over memory against passes with unpredictable branches.
LSD radix sort with d-bit digits makes $\lceil k/d \rceil$ passes over the array for k-bit keys, each a sequential read and a scatter of writes into 2^d output streams, with no data-dependent branches. Introsort makes $\Theta(\log n)$ partition passes; each reads sequentially from both ends of

its range, which is cache-friendly, but the comparison at every element is a branch that mispredicts about half the time on random keys, and the number of passes grows with n. We attribute radix sort's widening advantage above the crossing (Figure 2) to this difference in pass count and branch behaviour, and the flat floor below $\sim 10^3$ keys to the fixed cost of the 2^d-entry histograms. The structured families support the branch explanation directly: on nearly-sorted and reverse-sorted input introsort's comparisons become predictable and it runs 6–$11\times$ faster than on random keys of the same size, while radix sort, whose work does not depend on the key order, does not change. Whether radix sort's scattered stores are a bottleneck at large n is exactly what a counter run would show; the 16-bit-digit results, where a 512 KB histogram exceeds L1d and the advantage shrinks, suggest that the histogram's locality matters more than the output scatter.

Matrix multiplication: the access pattern of the base case.

Strassen and Karatsuba change the number of multiplications but not how the leaves access memory; their overhead is a constant number of extra additions and temporaries per level. This is consistent with crossings that are small and, on well-shaped inputs, insensitive to n above the cutoff, and with measured exponents within 0.04 of theory for both recursive algorithms at every cutoff. The i-j-k baseline is the clearest case of an access pattern, not an operation count, setting the time: its inner loop walks the second operand down a column with stride $8n$ bytes, so computing one row of the result touches every cache line of that operand; the operand is $8n^2$ bytes, which passes L1d (48 KB) at $n \approx 77$ and L2 (2 MB) at $n = 512$. The fitted exponent of 4.3 over $n \in [229{,}2048]$ is consistent with the operand leaving L2 within that range and every inner-loop step then paying a cache miss. We attribute the odd-size penalty of Strassen to the per-level padding copies, which the implementation makes explicitly and whose count depends on the number of odd dimensions met in the recursion.

Shortest paths: pointer chasing against arrays.

A binary heap of n entries is one contiguous array and a sift costs $\log_2 n$ steps whose addresses are computable in advance. A Fibonacci heap's consolidation walks a linked root list of $O(\log n)$ trees whose nodes were allocated in insertion order, each step a dependent load. Both do n delete-mins; the Fibonacci heap pays this cost on every one and, on sparse graphs, recoups little through decrease-keys because there are few: 0.05 per vertex at degree 2, 0.3 at degree 4 and on grids, 1.4 at degree 16 (operation counts from the harness log). On complete graphs there are 6 per vertex, and the ratio falls towards 1 as theory predicts. This is

the attribution Larkin, Sen and Tarjan (Larkin, Sen, and Tarjan 2014) made from measured L1 miss rates on a road network, and our timings are consistent with it; we have not measured it here.

6 Discussion

Crossovers are early, when they happen at all.
For three of the four pairs the asymptotically faster algorithm wins from inputs that fit comfortably in L1 cache: 1 500 keys, a 32×32 matrix, a 1 100-bit integer. The common belief that "the fancy algorithm only pays off for huge inputs" is, for these three, off by several orders of magnitude. What the belief gets right is the fourth pair: an algorithm whose advantage lies in an operation that realistic inputs rarely perform, and whose implementation trades array traversal for pointer chasing, can fail to win at any size that fits in memory.

The parameter moves the crossover more than the exponent does.
For Karatsuba the base-case cutoff shifts n^* by a factor of twenty (34 limbs at cutoff 32, 700 at cutoff 8) and halves the asymptotic advantage; for Strassen it moves n^* between 32 and roughly 1 000 and changes the speed at $n = 4096$ by $1.6\times$; the shape of the matrix—how many factors of two its dimension has—moves Strassen's crossing by a factor of ten; and the loop order of the *baseline* is worth $17\times$ at $n = 2048$ where Strassen's exponent is worth $2.6\times$. In the measured range, engineering decisions that the asymptotic analysis cannot see dominate the decision it can. The practical reading is that a recursive algorithm is only as good as its base-case policy, and that a "naive" baseline must be a competent one before any comparison means anything.

Wide intervals are answers, not failures.
Several crossings in Table 2 have intervals spanning a factor of two or more: Strassen with a too-small cutoff, Karatsuba with a too-small cutoff, radix sort with too-wide digits. In each the ratio curve approaches 1 slowly, and the bootstrap faithfully reports that the crossing could be anywhere in a wide range. For a practitioner this is the useful fact: over that range the two implementations are within noise of each other and the simpler one should be preferred for its simplicity; the asymptotic winner only earns its complexity beyond the top of the interval, and then only by the modest ratio we report there. Sharp crossings, by contrast, are where the decision is worth making carefully, and the sorting and Karatsuba (cutoff 32) rows show the grid can resolve them to $\pm 10\%$ and to a few limbs respectively.

Input structure can flip the verdict.
Radix sort's loss on nearly-sorted and reverse-sorted input and the Fibonacci heap's parity only on complete graphs are the same phenomenon seen from two sides. An algorithm with a worse bound but adaptive behaviour (introsort, the lazy heap) exploits structure that the input happens to have; an algorithm with a better bound and oblivious behaviour (radix sort, the Fibonacci heap's decrease-key) does exactly the same work regardless. Whether that is a feature or a bug depends on whether the structure is present, and the atlas gives the practitioner a way to check: the key is the input family, not the size.

Measured exponents are honest in the middle and lie at the edges.
Over the largest decade of each sweep the fitted exponents sit close to theory for the compute-bound algorithms (Karatsuba 1.55–1.60, schoolbook 1.97–2.01, Strassen 2.77–2.86) and drift above it for the memory-bound ones (introsort 1.14 on 64-bit keys where $n\log n$ predicts $\approx$ 1.06 over that decade; the i-j-k loop 4.3 where theory says 3). A fitted exponent on a log–log plot is a statement about the cache hierarchy as much as about the algorithm, which is why the cache boundaries are drawn on every runtime figure.

What the atlas cannot say.
These are single-threaded scalar implementations on one machine, measured under a hypervisor without hardware counters. A vectorised radix sort or a BLAS kernel would move every crossover, and not necessarily in the direction one expects, since vectorisation may help the simple kernel more than the recursive one built on it. The harness is offered as the way to redraw the map rather than the map as a permanent answer; Section 8 gives the one command that does so.

Guidance.
For a practitioner choosing between these pairs today, on hardware like ours:

- Sort with radix sort (11-bit digits) when keys are integers, inputs exceed about 1 500 elements, and the data is not already nearly or reverse sorted; use `std::sort` otherwise.
- Use Strassen with a base case of 32–64 for dense matrices above about 64 rows when the dimension is a multiple of a high power of two, or above about 300 otherwise (or pad once at the top)—and fix the loop order of the baseline first.
- Use Karatsuba above about 32 limbs with a cutoff of 32–128; the libraries already do.

- Do not use a Fibonacci heap for Dijkstra on any graph that fits in memory. Use a binary heap with lazy deletion unless the graph is a grid-like network, where an indexed binary heap is slightly better.

7 Threats to validity

Single machine.
Crossovers depend on cache sizes, memory bandwidth and the branch predictor. We report one machine in full and treat cross-machine generalisation as future work, offering the harness as the means.

Implementation quality.
Both sides are deliberately plain. A SIMD radix sort or a BLAS kernel would move every crossover, and so, to a smaller degree, would compiling with `-O3 -march=native`; the harness provides that variant (`make native-quick`) but we have not run it for this paper.

Virtualisation.
WSL2 adds timer and scheduling noise and hides counters. The interleaving and the bootstrap are designed to make the ratio robust to it, and the interval widths make the residual noise visible rather than hidden.

Input families.
Each family is a convenience sample of structures known to matter (sortedness, key width, density); the atlas is a map, not a census.

8 Reproducibility

All code, raw measurements, analysis scripts and figure sources are at https://github.com/dhanushp08/algorithm-crossover-atlas (tag `v1.0`; archived on Zenodo at 10.5281/zenodo.23809269). `make` builds and tests the code; `make full` and `make refine` re-measure the sweeps, and the analysis command in the README regenerates every number, table and figure in this paper from the raw measurements; `docs/BENCHMARKING.md` describes the machine setup. The raw data of the reported runs (`full`, `refine` and `patch`, Section 3.4, together with three preliminary `quick` runs) is in `results/`.

9 Conclusion

Asymptotic analysis answers “which algorithm wins eventually”; this paper answers “from what size” for four pairs that programmers choose between every day, on one well-described machine, with intervals rather than point guesses. Three of the four crossovers are small enough to fit in first-level cache and, for well-tuned configurations, sharp; one does not exist on any sparse input and reaches only parity on dense ones; and in every case the tuning parameter, the input structure or the shape of the problem moved the answer more than the exponent did. Where the data cannot pin a crossover down, we report the range rather than a point, because within that range the choice is immaterial. The harness that produced the atlas rebuilds it with one command, so the map can be redrawn as the hardware underneath it changes.

Declarations

Funding
The author received no funding for this work.

Competing interests
The author declares no competing interests.

Ethics approval and consent to participate
Not applicable. This study involved no human participants, animals or personal data.

Consent for publication
Not applicable.

Data availability
The raw timing measurements of the full, refine and patch runs (and of three preliminary quick runs), the processed results, and the analysis scripts that regenerate every number, table and figure in this paper are available in the GitHub repository https://github.com/dhanushp08/algorithm-crossover-atlas (tag v1.0) and are archived on Zenodo at 10.5281/zenodo.23809269.

Code availability
The C++17 implementations of all eight algorithms, the benchmarking harness, the randomized test suites, the benchmark and plotting scripts, and the documentation are available at https://github.com/dhanushp08/algorithm-crossover-atlas under the MIT License.

The exact version used for this paper is tag v1.0, archived on Zenodo at 10.5281/zenodo.23809269.

Use of generative AI tools

Claude Opus 5 (Anthropic) was used to assist with the literature review and with grammar and wording revisions. The author reviewed all AI-assisted text, verified every cited source, and takes full responsibility for the content of this manuscript (see Section 3.7).

Author contributions

Dhanush Paturu conceived the study, designed and implemented the algorithms and the benchmarking harness, performed the experiments, analysed the data, and wrote and revised the manuscript.

Appendix A: All crossover points

Table A1. All sustained crossover points, for every parameter value of every pair. Columns and notation as in Table 2; the 95% intervals are the shortest intervals over 1000 bootstrap resamples of the repetitions.

Pair	Input	Param	Comparison	n^*	95% CI	Outcome	$\rho(n_{\max})$
sorting	few unique	11	radix vs introsort	2.7 k	[2.4 k, 2.8 k]	crosses	0.51
sorting	few unique	16	radix vs introsort	7.5 k	[6.2 k, 8.1 k]	crosses	0.38
sorting	few unique	8	radix vs introsort	3.0 k	[2.7 k, 3.1 k]	crosses	0.71
sorting	nearly sorted	11	radix vs introsort	–	–	never faster to 67.1 M	3.69
sorting	nearly sorted	16	radix vs introsort	–	–	never faster to 67.1 M	2.49
sorting	nearly sorted	8	radix vs introsort	–	–	never faster to 67.1 M	4.52
sorting	organ pipe	11	radix vs introsort	1.6 k	[1.5 k, 1.8 k]	crosses	0.72
sorting	organ pipe	16	radix vs introsort	6.8 k	[6.6 k, 7.0 k]	crosses	0.45
sorting	organ pipe	8	radix vs introsort	≈48.1 M	[38.6 M, 55.7 M]	parity only	0.96
sorting	reverse	11	radix vs introsort	–	–	never faster to 67.1 M	8.23
sorting	reverse	16	radix vs introsort	–	–	never faster to 67.1 M	5.83
sorting	reverse	8	radix vs introsort	–	–	never faster to 67.1 M	10.28
sorting	uniform16	11	radix vs introsort	1.6 k	[1.6 k, 1.6 k]	crosses	0.25
sorting	uniform16	16	radix vs introsort	3.1 k	[3.0 k, 3.2 k]	crosses	0.37
sorting	uniform16	8	radix vs introsort	1.6 k	[1.6 k, 1.6 k]	crosses	0.27
sorting	uniform32	11	radix vs introsort	1.5 k	[1.4 k, 1.6 k]	crosses	0.25
sorting	uniform32	16	radix vs introsort	3.4 k	[3.1 k, 3.6 k]	crosses	0.37
sorting	uniform32	8	radix vs introsort	1.6 k	[1.4 k, 1.7 k]	crosses	0.24
sorting	uniform64	11	radix vs introsort	1.7 k	[1.6 k, 2.1 k]	crosses	0.37
sorting	uniform64	16	radix vs introsort	6.3 k–8.6 M	[6.3 k, 8.6 M]	crosses (gradual)	0.73
sorting	uniform64	8	radix vs introsort	1.7 k	[1.6 k, 2.1 k]	crosses	0.32
sorting	uniform8	11	radix vs introsort	1.4 k	[1.4 k, 1.5 k]	crosses	0.32
sorting	uniform8	16	radix vs introsort	3.5 k	[3.4 k, 3.6 k]	crosses	0.29
sorting	uniform8	8	radix vs introsort	1.4 k	[1.3 k, 1.4 k]	crosses	0.41
matmul	integer (2^k, 3·2^k)	128	Strassen vs naive i-k-j	≤ 256	–	faster at every n	0.41
matmul	integer (2^k, 3·2^k)	16	Strassen vs naive i-k-j	99–833	[99, 833]	crosses (gradual)	0.36

Pair	Input	Param	Comparison	n^*	95% CI	Outcome	$\rho(n_{\max})$
matmul	integer (2^k, 3·2^k)	256	Strassen vs naive i-k-j	≤ 256	–	faster at every n	0.43
matmul	integer (2^k, 3·2^k)	32	Strassen vs naive i-k-j	32–100	[32, 100]	crosses (gradual)	0.31
matmul	integer (2^k, 3·2^k)	64	Strassen vs naive i-k-j	69	[64, 76]	crosses	0.37
matmul	integer (2^k, 3·2^k)	8	Strassen vs naive i-k-j	811	[726, 853]	crosses	0.49
matmul	uniform (2^k, 3·2^k)	128	Strassen vs naive i-k-j	256–1.1 k	[256, 1.1 k]	crosses (gradual)	0.40
matmul	uniform (2^k, 3·2^k)	16	Strassen vs naive i-k-j	50–905	[50, 905]	crosses (gradual)	0.35
matmul	uniform (2^k, 3·2^k)	256	Strassen vs naive i-k-j	256–1.1 k	[256, 1.1 k]	crosses (gradual)	0.42
matmul	uniform (2^k, 3·2^k)	32	Strassen vs naive i-k-j	≤ 32	–	faster at every n	0.31
matmul	uniform (2^k, 3·2^k)	64	Strassen vs naive i-k-j	≤ 64	–	faster at every n	0.37
matmul	uniform (2^k, 3·2^k)	8	Strassen vs naive i-k-j	778–1.3 k	[778, 1.3 k]	crosses (gradual)	0.49
matmul	integer (all n)	128	Strassen vs naive i-k-j	239–840	[239, 840]	crosses (gradual)	0.41
matmul	integer (all n)	16	Strassen vs naive i-k-j	241–834	[241, 834]	crosses (gradual)	0.36
matmul	integer (all n)	256	Strassen vs naive i-k-j	≤ 256	–	faster at every n	0.43
matmul	integer (all n)	32	Strassen vs naive i-k-j	218	[202, 235]	crosses	0.31
matmul	integer (all n)	64	Strassen vs naive i-k-j	239	[233, 244]	crosses	0.37
matmul	integer (all n)	8	Strassen vs naive i-k-j	811	[723, 854]	crosses	0.49
matmul	uniform (all n)	128	Strassen vs naive i-k-j	313–1.1 k	[313, 1.1 k]	crosses (gradual)	0.40
matmul	uniform (all n)	16	Strassen vs naive i-k-j	443–890	[443, 890]	crosses (gradual)	0.35
matmul	uniform (all n)	256	Strassen vs naive i-k-j	256–1.1 k	[256, 1.1 k]	crosses (gradual)	0.42
matmul	uniform (all n)	32	Strassen vs naive i-k-j	297	[231, 308]	crosses	0.31
matmul	uniform (all n)	64	Strassen vs naive i-k-j	241	[235, 246]	crosses	0.37
matmul	uniform (all n)	8	Strassen vs naive i-k-j	778–1.3 k	[778, 1.3 k]	crosses (gradual)	0.49
matmul	integer (2^k, 3·2^k)	128	Strassen vs naive i-j-k	≤ 256	–	faster at every n	0.04
matmul	integer (2^k, 3·2^k)	16	Strassen vs naive i-j-k	106	[98, 121]	crosses	0.03
matmul	integer (2^k, 3·2^k)	256	Strassen vs naive i-j-k	≤ 256	–	faster at every n	0.04
matmul	integer (2^k, 3·2^k)	32	Strassen vs naive i-j-k	32–89	[32, 89]	crosses (gradual)	0.03
matmul	integer (2^k, 3·2^k)	64	Strassen vs naive i-j-k	71	[64, 77]	crosses	0.03
matmul	integer (2^k, 3·2^k)	8	Strassen vs naive i-j-k	150	[138, 159]	crosses	0.04
matmul	uniform (2^k, 3·2^k)	128	Strassen vs naive i-j-k	≤ 256	–	faster at every n	0.03
matmul	uniform (2^k, 3·2^k)	16	Strassen vs naive i-j-k	103	[99, 106]	crosses	0.03
matmul	uniform (2^k, 3·2^k)	256	Strassen vs naive i-j-k	≤ 256	–	faster at every n	0.03
matmul	uniform (2^k, 3·2^k)	32	Strassen vs naive i-j-k	≤ 32	–	faster at every n	0.02
matmul	uniform (2^k, 3·2^k)	64	Strassen vs naive i-j-k	72	[71, 75]	crosses	0.03
matmul	uniform (2^k,	8	Strassen vs naive i-	169	[158, 174]	crosses	0.04

Pair	Input	Param	Comparison	n^*	95% CI	Outcome	$\rho(n_{\max})$
	3·2^k)		j-k				
matmul	integer (all n)	128	Strassen vs naive i-j-k	≤ 136	–	faster at every n	0.04
matmul	integer (all n)	16	Strassen vs naive i-j-k	223	[217, 230]	crosses	0.03
matmul	integer (all n)	256	Strassen vs naive i-j-k	≤ 256	–	faster at every n	0.04
matmul	integer (all n)	32	Strassen vs naive i-j-k	32–89	[32, 89]	crosses (gradual)	0.03
matmul	integer (all n)	64	Strassen vs naive i-j-k	68–116	[68, 116]	crosses (gradual)	0.03
matmul	integer (all n)	8	Strassen vs naive i-j-k	243	[241, 247]	crosses	0.04
matmul	uniform (all n)	128	Strassen vs naive i-j-k	≤ 136	–	faster at every n	0.03
matmul	uniform (all n)	16	Strassen vs naive i-j-k	294	[286, 303]	crosses	0.03
matmul	uniform (all n)	256	Strassen vs naive i-j-k	≤ 256	–	faster at every n	0.03
matmul	uniform (all n)	32	Strassen vs naive i-j-k	59	[48, 69]	crosses	0.02
matmul	uniform (all n)	64	Strassen vs naive i-j-k	67–116	[67, 116]	crosses (gradual)	0.03
matmul	uniform (all n)	8	Strassen vs naive i-j-k	244	[242, 244]	crosses	0.04
bigint	ones	128	Karatsuba vs schoolbook	≤ 137	–	faster at every n	0.04
bigint	ones	16	Karatsuba vs schoolbook	35–60	[35, 60]	crosses (gradual)	0.03
bigint	ones	32	Karatsuba vs schoolbook	≤ 41	–	faster at every n	0.03
bigint	ones	64	Karatsuba vs schoolbook	≤ 91	–	faster at every n	0.03
bigint	ones	8	Karatsuba vs schoolbook	83–141	[83, 141]	crosses (gradual)	0.04
bigint	uniform	128	Karatsuba vs schoolbook	≤ 137	–	faster at every n	0.05
bigint	uniform	16	Karatsuba vs schoolbook	34–63	[34, 63]	crosses (gradual)	0.07
bigint	uniform	32	Karatsuba vs schoolbook	34	[32, 42]	crosses	0.05
bigint	uniform	64	Karatsuba vs schoolbook	65	[64, 70]	crosses	0.05
bigint	uniform	8	Karatsuba vs schoolbook	664–1.1 k	[664, 1.1 k]	crosses (gradual)	0.10
dijkstra	dense	-	Fib. heap vs binary (dec-key)	≈7.0 k	[6.5 k, 7.6 k]	parity only	0.98
dijkstra	grid	-	Fib. heap vs binary (dec-key)	–	–	never faster to 8.4 M	2.64
dijkstra	random	16	Fib. heap vs binary (dec-key)	–	–	never faster to 8.4 M	1.41
dijkstra	random	2	Fib. heap vs binary (dec-key)	–	–	never faster to 8.4 M	2.09
dijkstra	random	256	Fib. heap vs binary (dec-key)	–	–	never faster to 1.0 M	1.29
dijkstra	random	4	Fib. heap vs binary (dec-key)	–	–	never faster to 8.4 M	1.73
dijkstra	random	64	Fib. heap vs binary (dec-key)	–	–	never faster to 4.2 M	1.30
dijkstra	dense	-	Fib. heap vs binary (lazy)	–	–	never faster to 7.7 k	1.73
dijkstra	grid	-	Fib. heap vs binary (lazy)	–	–	never faster to 8.4 M	2.19
dijkstra	random	16	Fib. heap vs binary (lazy)	–	–	never faster to 8.4 M	1.52
dijkstra	random	2	Fib. heap vs binary (lazy)	–	–	never faster to 8.4 M	2.52

Pair	Input	Param	Comparison	n^*	95% CI	Outcome	$\rho(n_{\max})$
dijkstra	random	256	Fib. heap vs binary (lazy)	–	–	never faster to 1.0 M	1.35
dijkstra	random	4	Fib. heap vs binary (lazy)	–	–	never faster to 8.4 M	2.38
dijkstra	random	64	Fib. heap vs binary (lazy)	–	–	never faster to 4.2 M	1.22

References

Alman, Josh, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. 2025. “More Asymmetry Yields Faster Matrix Multiplication.” In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005–39. SIAM. https://doi.org/10.1137/1.9781611978322.63.

Alman, Josh, and Virginia Vassilevska Williams. 2024. “A Refined Laser Method and Faster Matrix Multiplication.” *TheoretiCS* 3. https://doi.org/10.46298/theoretics.24.21.

Benson, Austin R., and Grey Ballard. 2015. “A Framework for Practical Parallel Fast Matrix Multiplication.” In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*, 42–53. ACM. https://doi.org/10.1145/2688500.2688513.

Bernstein, Daniel J. 2001. “Multidigit Multiplication for Mathematicians.” https://cr.yp.to/papers.html#m3.

Brodal, Gerth Stølting, George Lagogiannis, and Robert E. Tarjan. 2012. “Strict Fibonacci Heaps.” In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC ’12)*, 1177–84. ACM. https://doi.org/10.1145/2213977.2214082.

Chen, Mo, Rezaul Alam Chowdhury, Vijaya Ramachandran, David Lan Roche, and Lingling Tong. 2007. “Priority Queues and Dijkstra’s Algorithm.” TR-07-54. Department of Computer Sciences, The University of Texas at Austin. https://www.cs.utexas.edu/ftp/techreports/tr07-54.pdf.

Cherkassky, Boris V., Andrew V. Goldberg, and Tomasz Radzik. 1996. “Shortest Paths Algorithms: Theory and Experimental Evaluation.” *Mathematical Programming* 73 (2): 129–74. https://doi.org/10.1007/BF02592101.

Curtsinger, Charlie, and Emery D. Berger. 2013. “STABILIZER: Statistically Sound Performance Evaluation.” In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’13)*, 219–28. ACM. https://doi.org/10.1145/2451116.2451141.

D’Alberto, Paolo, and Alexandru Nicolau. 2007. “Adaptive Strassen’s Matrix Multiplication.” In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS ’07)*, 284–92. ACM. https://doi.org/10.1145/1274971.1275010.

Drepper, Ulrich. 2007. “What Every Programmer Should Know about Memory.” Red Hat, Inc. https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

Fog, Agner. 2026. *Optimizing Software in C++: An Optimization Guide for Windows, Linux and Mac Platforms*. Last updated September 10, 2026. https://www.agner.org/optimize/optimizing_cpp.pdf.

Fredman, Michael L., and Robert Endre Tarjan. 1987. “Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms.” *Journal of the ACM* 34 (3): 596–615. https://doi.org/10.1145/28869.28874.

Granlund, Torbjörn, and the GMP development team. 2022. “GMP Repository: Alder Lake Parameter File, `mpn/x86_64/alderlake/gmp-mparam.h`.” gmp-commit mailing list, changeset of March 2022. https://gmplib.org/list-archives/gmp-commit/2022-March/003156.html.

Granlund, Torbjörn, and the GMP development team. 2023. *GNU MP: The GNU Multiple Precision Arithmetic Library*, Section 15.1.2 “Karatsuba Multiplication.” 6.3.0 ed. Free Software Foundation. https://gmplib.org/manual/Karatsuba-Multiplication.

Harvey, David, and Joris van der Hoeven. 2021. “Integer Multiplication in Time $O(n\log n)$.” *Annals of Mathematics* 193 (2): 563–617. https://doi.org/10.4007/annals.2021.193.2.4.

Hoefler, Torsten, and Roberto Belli. 2015. “Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results.” In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’15)*, 73:1–12. ACM. https://doi.org/10.1145/2807591.2807644.

Huang, Jianyu, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. 2016. “Strassen’s Algorithm Reloaded.” In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, 690–701. IEEE. https://doi.org/10.1109/SC.2016.58. Extended version: “Implementing Strassen’s Algorithm with BLIS,” FLAME Working Note #79, https://arxiv.org/abs/1605.01078.

Huss-Lederman, Steven, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Turnbull. 1996. “Implementation of Strassen’s Algorithm for Matrix Multiplication.” In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, 32. IEEE Computer Society. https://doi.org/10.1145/369028.369096.

Johnson, David S. 2002. “A Theoretician’s Guide to the Experimental Analysis of Algorithms.” In *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, edited by Michael H. Goldwasser, David S. Johnson, and Catherine C. McGeoch, 215–50. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 59. Providence, RI: American Mathematical Society. https://doi.org/10.1090/dimacs/059/11.

Karatsuba, Anatolii, and Yuri Ofman. 1962. “Multiplication of Many-Digital Numbers by Automatic Computers.” *Doklady Akademii Nauk SSSR* 145 (2): 293–94. https://geodesic.mathdoc.fr/item/DAN_1962_145_2_a17.

LaMarca, Anthony, and Richard E. Ladner. 1999. “The Influence of Caches on the Performance of Sorting.” *Journal of Algorithms* 31 (1): 66–104. https://doi.org/10.1006/jagm.1998.0985.

Larkin, Daniel H., Siddhartha Sen, and Robert E. Tarjan. 2014. “A Back-to-Basics Empirical Study of Priority Queues.” In *Proceedings of the 16th Workshop on Algorithm Engineering and Experiments (ALENEX 2014)*, 61–72. SIAM. https://doi.org/10.1137/1.9781611973198.7.

Linux man-pages project. 2026. *perf_event_open(2) – Set up Performance Monitoring*. The Linux man-pages project. https://www.man7.org/linux/man-pages/man2/perf_event_open.2.html.

Lipton, Richard J. 2010. “Galactic Algorithms.” Gödel’s Lost Letter and P=NP (blog), October 23. https://rjlipton.com/2010/10/23/galactic-algorithms/.

McGeoch, Catherine C. 2012. *A Guide to Experimental Algorithmics*. Cambridge, UK: Cambridge University Press. https://doi.org/10.1017/CBO9780511843747.

Musser, David R. 1997. “Introspective Sorting and Selection Algorithms.” *Software: Practice and Experience* 27 (8): 983–93. https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#.

Mytkowicz, Todd, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. “Producing Wrong Data Without Doing Anything Obviously Wrong!” In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, 265–76. ACM. https://doi.org/10.1145/1508244.1508275.

Polychroniou, Orestis, and Kenneth A. Ross. 2014. “A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison- and Radix-Sort.” In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 755–66. ACM. https://doi.org/10.1145/2588555.2610522.

Rahman, Naila, and Rajeev Raman. 2001. “Adapting Radix Sort to the Memory Hierarchy.” *ACM Journal of Experimental Algorithmics* 6: Article 7. https://doi.org/10.1145/945394.945401.

Sanders, Peter. 2009. “Algorithm Engineering – an Attempt at a Definition.” In *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, edited by Susanne Albers, Helmut Alt, and Stefan Näher. Vol. 5760. Lecture Notes in Computer Science, 321–40. Springer. https://doi.org/10.1007/978-3-642-03456-5_22.

Satish, Nadathur, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. “Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort.” In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 351–62. ACM. https://doi.org/10.1145/1807167.1807207.

Strassen, Volker. 1969. “Gaussian Elimination Is Not Optimal.” *Numerische Mathematik* 13 (4): 354–56. https://doi.org/10.1007/BF02165411.