Note that with 17,576 buckets, HASH SORT outperforms QUICKSORT for n> 8,192 items (and this trend continues with increasing n). However, with only 676 buckets, once n>32,768 (for an average of 48 elements per bucket), HASH SORT begins its inevitable slowdown with the accumulated cost of executing INSERTION SORT on increasingly larger sets. Indeed, with only 26 buckets, once n>256, HASH SORT begins to quadruple its performance as the problem size doubles, showing how too few buckets leads to $O(n^2)$ performance.

Criteria for Choosing a Sorting Algorithm

To choose a sorting algorithm, consider the qualitative criteria in Table 4-6. These may help your initial decision, but you likely will need more quantitative measures to guide you.

Table 4-6. Criteria for choosing a sorting algorithm

Criteria	Sorting algorithm
Only a few items	INSERTION SORT
Items are mostly sorted already	INSERTION SORT
Concerned about worst-case scenarios	HEAP SORT
Interested in a good average-case result	QUICKSORT
Items are drawn from a dense universe	BUCKET SORT
Desire to write as little code as possible	INSERTION SORT

To choose the appropriate algorithm for different data, you need to know some properties about your input data. We created several benchmark data sets on which to show how the algorithms presented in this chapter compare with one another. Note that the actual values of the generated tables are less important because they reflect the specific hardware on which the benchmarks were run. Instead, you should pay attention to the relative performance of the algorithms on the corresponding data sets:

Random strings
: Throughout this chapter, we have demonstrated performance of sorting algorithms when sorting 26-character strings that are permutations of the letters in the alphabet. Given there are $n!$ such strings, or roughly $4.03*10^{26}$ strings, there are few duplicate strings in our sample data sets. In addition, the cost of comparing elements is not constant, because of the occasional need to compare multiple characters.

Double precision floating-point values
: Using available pseudorandom generators available on most operating systems, we generate a set of random numbers from the range [0,1). There are essentially no duplicate values in the sample data set and the cost of comparing two elements is a fixed constant.

The input data provided to the sorting algorithms can be preprocessed to ensure some of the following properties (not all are compatible):

Sorted
: The input elements can be presorted into ascending order (the ultimate goal) or in descending order.

Killer median-of-three
: Musser (1997) discovered an ordering that ensures that QUICKSORT requires $O(n^2)$ comparisons when using median-of-three to choose a pivot.

Nearly sorted
: Given a set of sorted data, we can select k pairs of elements to swap and the distance d with which to swap (or 0 if any two pairs can be swapped). Using this capability, you can construct input sets that might more closely match your input set.

The upcoming tables are ordered left to right, based upon how well the algorithms perform on the final row in the table. Each section has four tables, showing performance results under the four different situations outlined earlier in this chapter.

String Benchmark Results

Because INSERTION SORT and SELECTION SORT are the two slowest algorithms in this chapter on randomly uniform data (by several orders of magnitude) we omit these algorithms from Tables 4-7 through 4-11. However, it is worth repeating that on sorted data (Table 4-8) and nearly sorted data (Tables 4-10 and 4-11) INSERTION SORT will outperform the other algorithms, often by an order of magnitude. To produce the results shown in Tables 4-7 through 4-11, we executed each trial 100 times on the high-end computer and discarded the best and worst performers. The average of the remaining 98 trials is shown in these tables. The columns labeled QUICKSORT BFPRT4 MINSIZE=4 refer to a QUICKSORT implementation that uses BFPRT (with groups of 4) to select the partition value and which switches to INSERTION SORT when a subarray to be sorted has four or fewer elements.

Table 4-7. Performance results (in seconds) on random 26-letter permutations of the alphabet

n	Hash Sort 17,576 buckets	Quicksort median-of-three	Heap Sort	Median Sort	Quicksort BFPRT4 minSize=4
4,096	0.0012	0.0011	0.0013	0.0023	0.0041
8,192	0.002	0.0024	0.0031	0.005	0.0096
16,384	0.0044	0.0056	0.0073	0.0112	0.022
32,768	0.0103	0.014	0.0218	0.0281	0.0556
65,536	0.0241	0.0342	0.0649	0.0708	0.1429
131,072	0.0534	0.0814	0.1748	0.1748	0.359

Table 4-8. Performance (in seconds) on ordered random 26-letter permutations of the alphabet

n	Hash Sort 17,576 buckets	Quicksort median-of-three	Heap Sort	Median Sort	Quicksort BFPRT4 minSize=4
4,096	0.0011	0.0007	0.0012	0.002	0.0031
8,192	0.0019	0.0015	0.0027	0.0042	0.007

Table 4-8. Performance (in seconds) on ordered random 26-letter permutations of the alphabet (continued)

n	Hash Sort 17,576 buckets	Quicksort median-of-three	Heap Sort	Median Sort	Quicksort BFPRT[4] minSize=4
16,384	0.0037	0.0036	0.0062	0.0094	0.0161
32,768	0.0074	0.0082	0.0157	0.0216	0.0381
65,536	0.0161	0.0184	0.0369	0.049	0.0873
131,072	0.0348	0.0406	0.0809	0.1105	0.2001

Table 4-9. Performance (in seconds) on killer median data

n	Hash Sort 17,576 buckets	Heap Sort	Median Sort	Quicksort BFPRT[4] minSize=4	Quicksort median-of-three[a]
4,096	0.0011	0.0012	0.0021	0.0039	0.0473
8,192	0.0019	0.0028	0.0045	0.0087	0.1993
16,384	0.0038	0.0066	0.0101	0.0194	0.8542
32,768	0.0077	0.0179	0.024	0.0472	4.083
65,536	0.0171	0.0439	0.056	0.1127	17.1604
131,072	0.038	0.1004	0.1292	0.2646	77.4519

[a] Because the performance of QUICKSORT median-of-three degrades so quickly, only 10 trials were executed; the table shows the average of eight runs once the best and worst performers were discarded.

Table 4-10. Performance (in seconds) on 16 random pairs of elements swapped eight locations away

n	Hash Sort 17,576 buckets	Quicksort median-of-three	Heap Sort	Median Sort	Quicksort BFPRT[4] minSize=4
4,096	0.0011	0.0007	0.0012	0.002	0.0031
8,192	0.0019	0.0015	0.0027	0.0042	0.007
16,384	0.0038	0.0035	0.0063	0.0094	0.0161
32,768	0.0072	0.0081	0.0155	0.0216	0.038
65,536	0.0151	0.0182	0.0364	0.0491	0.0871
131,072	0.0332	0.0402	0.08	0.1108	0.2015

Table 4-11. Performance (in seconds) on n/4 random pairs of elements swapped four locations away

n	Hash Sort 17,576 buckets	Quicksort median-of-three	Heap Sort	Median Sort	Quicksort BFPRT[4] minSize=4
4,096	0.0011	0.0008	0.0012	0.002	0.0035
8,192	0.0019	0.0019	0.0028	0.0044	0.0078
16,384	0.0039	0.0044	0.0064	0.0096	0.0175
32,768	0.0073	0.01	0.0162	0.0221	0.0417

Table 4-11. Performance (in seconds) on n/4 random pairs of elements swapped four locations away (continued)

n	Hash Sort 17,576 buckets	Quicksort median-of-three	Heap Sort	Median Sort	Quicksort BFPRT[4] minSize=4
65,536	0.0151	0.024	0.0374	0.0505	0.0979
131,072	0.0333	0.0618	0.0816	0.1126	0.2257

Double Benchmark Results

The benchmarks using double floating-point values (Tables 4-12 through 4-16) eliminate much of the overhead that was simply associated with string comparisons. Once again, we omit INSERTION SORT and SELECTION SORT from these tables.

Table 4-12. Performance (in seconds) on random floating-point values

n	Bucket Sort	Quicksort median-of-three	Median Sort	Heap Sort	Quicksort BFPRT[4] minSize=4
4,096	0.0009	0.0009	0.0017	0.0012	0.0003
8,192	0.0017	0.002	0.0039	0.0029	0.0069
16,384	0.0041	0.0043	0.0084	0.0065	0.0157
32,768	0.0101	0.0106	0.0196	0.0173	0.039
65,536	0.0247	0.0268	0.0512	0.0527	0.1019
131,072	0.0543	0.0678	0.1354	0.1477	0.26623

Table 4-13. Performance (in seconds) on ordered floating-point values

n	Bucket Sort	Heap Sort	Median Sort	Quicksort median-of-three	Quicksort BFPRT[4] minSize=4
4,096	0.0007	0.0011	0.0015	0.0012	0.0018
8,192	0.0015	0.0024	0.0032	0.0025	0.004
16,384	0.0035	0.0052	0.0067	0.0055	0.0089
32,768	0.0073	0.0127	0.015	0.0133	0.0208
65,536	0.0145	0.0299	0.0336	0.0306	0.0483
131,072	0.0291	0.065	0.0737	0.0823	0.1113

Table 4-14. Performance (in seconds) on killer median data

n	Bucket Sort	Heap Sort	Median Sort	Quicksort median-of-three	Quicksort BFPRT[4] minSize=4
4,096	0.0008	0.0011	0.0015	0.0015	0.0025
8,192	0.0016	0.0024	0.0034	0.0033	0.0056
16,384	0.0035	0.0053	0.0071	0.0076	0.0122

Table 4-14. Performance (in seconds) on killer median data (continued)

n	Bucket Sort	Heap Sort	Median Sort	Quicksort median-of-three	Quicksort BFPRT[4] minSize=4
32,768	0.0079	0.0134	0.0164	0.0192	0.0286
65,536	0.0157	0.0356	0.0376	0.0527	0.0686
131,072	0.0315	0.0816	0.0854	0.1281	0.1599

Table 4-15. Performance (in seconds) on 16 random pairs of elements swapped eight locations away

n	Bucket Sort	Heap Sort	Median Sort	Quicksort median-of-three	Quicksort BFPRT[4] minSize=4
4,096	0.0007	0.0011	0.0015	0.0012	0.0018
8,192	0.0015	0.0024	0.0032	0.0025	0.004
16,384	0.0035	0.0051	0.0067	0.0054	0.0089
32,768	0.0071	0.0127	0.0151	0.0133	0.0209
65,536	0.0142	0.0299	0.0336	0.0306	0.0482
131,072	0.0284	0.065	0.0744	0.0825	0.111

Table 4-16. Performance (in seconds) on n/4 random pairs of elements swapped four locations away

n	Bucket Sort	Heap Sort	Quicksort median-of-three	Median Sort	Quicksort BFPRT[4] minSize=4
4,096	0.0001	0.0014	0.0015	0.0019	0.005
8,192	0.0022	0.0035	0.0032	0.0052	0.012
16,384	0.0056	0.0083	0.0079	0.0099	0.0264
32,768	0.0118	0.0189	0.0189	0.0248	0.0593
65,536	0.0238	0.0476	0.045	0.0534	0.129
131,072	0.0464	0.1038	0.1065	0.1152	0.2754

References

Bentley, Jon Louis and M. Douglas McIlroy, "Engineering a Sort Function," *Software—Practice and Experience*, 23(11): 1249–1265, 1993, *http://citeseer.ist.psu.edu/bentley93engineering.html*.

Blum, Manuel, Robert Floyd, Vaughan Pratt, Ronald Rivest, and Robert Tarjan, "Time bounds for selection." *Journal of Computer and System Sciences*, 7(4): 448–461, 1973.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Second Edition. McGraw-Hill, 2001.