Sorting is one of the fundamental operations we will study in this course. The need to sort data has been critical since the inception of computer science. For example, bubble sort, one of the original sorting algorithms, was analyzed to have average and worst case performance of $O(n^2)$ in 1956. Although we know that comparison sorts cannot perform better than $O(n \log n)$ in the worst case, better performance is often possible—although not guaranteed—on real-world data. Because of this, new sorting algorithms continue to be proposed.

We start with an overview of sorting collections of records that can be stored entirely in memory. These approaches form the foundation for sorting very large data collections that must remain on disk.

### 7.1 Shell Sort

Recall our description of insertion sort used as an example in the order notation discussion. Here, we take an array $A[1 \ldots n]$ and sort it from left to right element by element.
Chapter 7. Sorting

element. Assuming $A[1]$ through $A[m-1]$ are sorted, we “insert” $A[m]$ into place by backing it up to its proper position in the partially sorted list. The average and worst case performance for insertion sort are both $O(n^2)$.

Shell sort was proposed by Donald L. Shell in 1956. Similar to insertion sort, it takes an element $A[m]$ and compares it to previous elements to push it into place. Rather than comparing with the neighbour $A[m-1]$ immediately to the left, however, we use larger increments of size $h > 1$ to generate sorted subgroups. We then slowly decrease the increment size until $h = 1$, at which point the array will be sorted. Because of this, Shell sort is sometimes referred to as diminishing insertion sort.

As an example, suppose we have a 16-element array $A[0 \ldots 15]$ and we divide it into eight groups with elements separated by a $h = 8$ increment.

\[(A_0, A_8), (A_1, A_9), \ldots, (A_7, A_{15}) \quad (7.1)\]

We sort each group independently using insertion sort. After we’re done, we know that $A_0 \leq A_8$, $A_1 \leq A_9$, $A_7 \leq A_{15}$. Now, we reduce the increment to $h = 4$, producing groups of size four.

\[(A_0, A_4, A_8, A_{12}), \ldots, (A_3, A_7, A_{11}, A_{15}) \quad (7.2)\]

We again apply insertion sort to sort each of these subgroups. Notice something important, however. Because of the previous step with $h = 8$, each subgroup is already partially sorted. For example, in the first subgroup we know that $A_0 \leq A_8$ and $A_4 \leq A_{12}$. This means that when we push $A_8$ into place, it may swap with $A_4$, but it will never swap with $A_0$ because it’s already “in place” relative to $A_0$.

This shows that the previous sorting step reduces the number of comparisons needed to complete the current sorting step. We continue to sort subgroups and reduce the increment down to $h = 1$. At this point, the array is already almost sorted due to the $h = 2$ sort step (that is, we know $A_0 \leq A_2$, $A_1 \leq A_3$, ..., $A_{13} \leq A_{15}$). This means the final insertion sort should run much faster than an insertion sort on the array in its original configuration.

The question for Shell sort is: Do the multiple insertion sorts with increments $h > 1$ cost more than the savings we achieve when we perform the final insertion sort with $h = 1$? It turns out we save more on the final insertion sort than it costs to configure the array for that sort, so Shell sort is on average faster than insertion sort.

Consider the simplest generalization. We will perform two steps, one with $h = 2$ followed by another with $h = 1$. After the $h = 2$ sort we have a 2-sorted array. Let’s analyze the performance of the final step, placing the 2-sorted array into order.

How many permutations $p$ are there for $n$ values such that $A_i \leq A_{i+2}$ for $0 \leq i \leq n-3$, that is, $n$ values that are 2-sorted? $p$ is $n$ choose $\lfloor n/2 \rfloor$, or more formally:

\[p = \binom{n}{\lfloor n/2 \rfloor} \quad (7.3)\]

Each permutation is equally likely. For any permutation, how many swaps are required to go from 2-sorted to “in order?” Let’s define $A_n$ to be the total number of
swaps needed to put all possible 2-sorted lists with \( n \) values in order. We can hand-compute this value for small \( n \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>2-Sorted Permutations</th>
<th>Swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1}</td>
<td>( A_1 = 0 )</td>
</tr>
<tr>
<td>2</td>
<td>{12, 21}</td>
<td>( A_2 = 0 + 1 = 1 )</td>
</tr>
<tr>
<td>3</td>
<td>{123, 132, 213}</td>
<td>( A_3 = 0 + 1 + 1 = 2 )</td>
</tr>
<tr>
<td>4</td>
<td>{1324, 1234, 1243, 2134, 2143, 3142}</td>
<td>( A_4 = 1 + 0 + 1 + 1 + 2 + 3 = 8 )</td>
</tr>
</tbody>
</table>

It turns out that the total number of swaps for a given \( n \) is \( A_n = \left\lfloor \frac{n}{2} \right\rfloor 2^{n-2} \). Therefore, the average number of swaps for any given permutation is

\[
A_n = \frac{\left\lfloor \frac{n}{2} \right\rfloor 2^{n-2}}{p} = \frac{\left\lfloor \frac{n}{2} \right\rfloor 2^{n-2}}{\left\lfloor \frac{n}{2} \right\rfloor} \approx 0.15n^{3/2}
\] (7.4)

This means the final insertion sort on a 2-sorted array runs in \( O(n^{3/2}) \) time, which is better than \( O(n^2) \). This is the savings we achieve if we first perform an insertion sort with \( h = 2 \) to produce the 2-sorted array.

How much does it cost to do the \( h = 2 \) sort step? We will not provide the details here, but it can be proven that 2-sorting an array of size \( n \) requires \( O(n^{3/2}) \) time. This means the overall cost of Shell sort with two steps is \( O(n^{3/2}) \), which is still better than insertion sort’s \( O(n^2) \) average case performance.

More steps that use larger increments add some additional improvements. The results will always fall somewhere between \( O(n^{3/2}) \) and \( O(n^2) \), that is, Shell sort is never worse than insertion sort, but a careful choice of \( h \) must be made at each step to see improved performance.

What is a “good” \( h \)? No theoretical answer to this question is known, but empirical evidence suggests

\[
\begin{align*}
  h_1 &= 1 \\
  h_2 &= 3h_1 + 1 = 4 \\
  h_3 &= 3h_2 + 1 = 13 \\
  &\vdots \\
  h_{s+1} &= 3h_s + 1 \\
\end{align*}
\] (7.5)

We stop when \( h_{s+2} \geq n \). A little math shows this is equivalent to \( h_s = \left\lfloor \frac{n}{2^s} \right\rfloor \). This is the increment we use for the first insertion sort step.
7.2 Quicksort

Quicksort was originally proposed by C.A.R. Hoare in 1962. In spite of its age, it is still used as a standard method of sorting. Quicksort runs in $O(n \log n)$ average time by applying a “divide and conquer” approach.

1. Split the array into partitions about a pivot. Values in the left partition are less than the pivot. Values in the right partition are greater than or equal to the pivot.

2. Recursively sort the two parts.

A partitioning function is used to take a subset of an array $A$ and split its value about a pivot element $x$. The left partition contains elements $< x$, and the right partition contains elements $\geq x$.

```
partition(A, lf, rt, lp, rp)
Input: A[], array to sort; lf, left boundary; rt, right boundary; lp, end of left partition; rp, start of right partition
i = lf+1 // Start of right partition
j = lf+1 // Elem to position
x = A[lf] // Pivot
while j ≤ rt do
    if A[j] < x then
        i++
    end
    j++
end
A[i-1] = x
lp = i-2 // Set end of left partition
rp = i // Set start of right partition
```

Any element in the subset can act as a pivot. Our pseudocode uses the first element in the subset, although we’ll see later that this is usually not a good choice.

Partitioning occurs by walking through elements $A[j]$ in $A$’s subset one-by-one, $j = lf+1, \ldots, rt$, and placing them in the proper partition (Fig. 7.2a). Each element $A[j]$ is compared to the pivot $x = A[lf]$. There are two possible cases.


After processing all the elements in the subset of $A$, both regions will be complete (Fig. 7.2b). The final step is to move the pivot between the two regions. This is done by swapping $A[lf]$ with $A[i-1]$ (Fig. 7.2c). Table 7.1 shows a complete example of partitioning an array $A$ with eight elements.
7.2. Quicksort

![Diagram of Quicksort partitioning](image)

Figure 7.2. Different stages of Quicksort partitioning: (a) during partitioning; (b) after partitioning is complete; (c) after the pivot is moved and partition boundaries are set

<table>
<thead>
<tr>
<th>Step</th>
<th>A[lf...rt]</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>17 32 14 16 22 18 44 68</td>
<td>rt</td>
</tr>
<tr>
<td>1</td>
<td>17 32 14 16 22 18 44 68</td>
<td>rt</td>
</tr>
<tr>
<td>2</td>
<td>17 32 14 16 22 18 44 68</td>
<td>A[j] ≥ x, j++</td>
</tr>
<tr>
<td>3</td>
<td>17 32 14 16 22 18 44 68</td>
<td>A[j] ≥ x, i++, j++</td>
</tr>
<tr>
<td>4</td>
<td>17 32 14 16 22 18 44 68</td>
<td>A[j] ≥ x, j++</td>
</tr>
<tr>
<td>5</td>
<td>17 32 14 16 22 18 44 68</td>
<td>A[j] ≥ x, j++</td>
</tr>
<tr>
<td>6</td>
<td>17 32 14 16 22 18 44 68</td>
<td>A[j] ≥ x, j++</td>
</tr>
</tbody>
</table>

Table 7.1. An example of partitioning an array during Quicksort

Given the partition function, we use it to split A into a left and right partition, then recursively sort the partitions. The recursion continues until the size of a partition is 1 or smaller. Partitions of this size are, by default, in sorted order.
Chapter 7. Sorting

![Recursive Partition Trees](image)

Figure 7.3. Examples of recursive partition trees: (a) best case balanced partitioning; (b) worst case unbalanced partitioning

```c
qsort(A, lf, rt)
Input: A[], array to sort; rt, right boundary
if lf < rt then
    partition( A, lf, rt, lp, rp ) // Partition subarray
    qsort( A, lf, lp ) // Recursively sort the partitions
    qsort( A, rp, rt )
end
```

In order to sort an array $A$ of size $n$, we call $\text{qsort}( A, 0, n-1 )$.

**Best Case Performance.** In the best case Quicksort performs balanced partitions, splitting a collection of size $n$ into two parts that are roughly size $n/2$, then splitting those parts into four parts of size $n/4$, and so on until we get $n$ parts of size 1 (Fig 7.3a). The amount of work required to partition each level in the recursion is $O(n)$: one partition of size $n$, then two partitions of size $n/2$, four partitions of size $n/4$, and so on.

If a collection of size $n$ is repeatedly halved, it takes $\log_2 n = \lg n$ divisions to reduce the size to 1. So there are $\lg n$ levels of recursion, each requiring $O(n)$ work to partition, for a total of $O(n \lg n)$.

**Worst Case Performance.** If Quicksort produces a completely unbalanced partition, we will only remove one element—the pivot—from the collection on each recursive step. This produces $n$ recursive levels of size $n$, $n-1$, $n-2$, down to 1 element. The work required to partition all of these levels is $\sum_{i=1}^{n} i = \frac{1}{2}n^2 + \frac{1}{2}n$, which is $O(n^2)$.

**Average Case Performance.** Quicksort’s average case performance depends on what we define as an “average case.” Suppose Quicksort partitions each array or subarray consistently, but not in a fully balanced way. For example, suppose Quicksort places 10\% of the elements in the left partition and 90\% of the elements in the right partition every time it splits. A little math shows this runs in $O(n \log \frac{n}{2})$ time, that is, still in $n \log n$ time but with a smaller base for the logarithm.
7.3. Heapsort

Heapsort is another common sorting algorithm. Unlike Quicksort, heapsort guarantees \(O(n \log n)\) performance, even in the worst case. In absolute terms, however, heapsort is slower than Quicksort. If the possibility of a worst case \(O(n^2)\) is not acceptable, heapsort would normally be chosen over Quicksort.

Heapsort works in a manner similar to a tournament sort. In a tournament sort all pairs of values are compared and a “winner” is promoted to the next level of the tournament. Successive winners are compared and promoted until a single overall winner is found. Tournament sorting a set of numbers where the bigger number wins identifies the largest number in the collection (Fig. 7.4). Once a winner is found, we re-evaluate its winning path to promote a second winner, then a third winner, and so on until all the numbers are promoted, returning the collection in reverse sorted order.

It takes \(O(n)\) time to build the initial tournament structure and promote the first element. Re-evaluating a winning path requires \(O(\log n)\) time, since the height of the tournament tree is \(\log n\). Promoting all \(n\) values therefore requires \(O(n \log n)\) time. The main drawback of tournament sort is that it needs about \(2n\) space to sort a collection of size \(n\).

Heapsort can sort in-place in the original array. To being, we define a heap as an array \(A[1 \ldots n]\) that satisfies the following rule\(^1\).

\[
\begin{align*}
\end{align*}
\] (7.6)

To sort in place, heapsort splits \(A\) into two parts: a heap at the front of \(A\), and a partially sorted list at the end of \(A\). As elements are promoted to the front of the heap, they are swapped with the element at the end of the heap. This grows the sorted list.

\(^1\)Note that heaps are indexed starting at 1, not at 0 like a C array. The heapsort algorithms will not work unless the array starts at index 1.
Chapter 7. Sorting

list and shrinks the heap until the entire collection is sorted. Specifically, heapsort executes the following steps.

1. Manipulate $A$ into a heap.
5. Continue re-adjusting and swapping until the heap is empty and the partially sorted list contains all $n$ elements in $A$.

We first describe how to perform the third step: re-adjusting $A$ to ensure it satisfies the heap property. Since we started with a valid heap, the only element that might be out of place is $A[1]$. The following sift algorithm pushes an element $A[i]$ at position $i$ into a valid position, while ensuring no other elements are moved in ways that violate the heap property (Fig. 7.5b,c).

$sift(A, i, n)$
Input: $A[\ ]$, heap to correct; $i$, element possibly out of position; $n$, size of heap

while $i \leq \lfloor n/2 \rfloor$ do
    $j = i \times 2$ // $j = 2i$
    $k = j + 1$ // $k = 2i + 1$
    if $k \leq n \text{ and } A[k] \geq A[j]$ then
    else
    end
    if $A[i] \geq A[lg]$ then
    end
    $i = lg$
end

So, to move $A[1]$ into place after swapping, we would call $sift(A, 1, n-1)$. Notice that $sift$ isn’t specific to $A[1]$. It can be used to move any element into place. This allows us to use $sift$ to convert $A$ from its initial configuration into a heap.

$heapify(A, n)$
Input: $A[\ ]$, array to heapify; $n$, size of array

$i = \lfloor n/2 \rfloor$
while $i \geq 1$ do
    $sift(A, i, n)$ // Sift $A[i]$ to satisfy heap constraints
    $i--$
end
7.3. Heapsort

The heapify function assumes the rightmost element that might violate the heap constraints is \( A[\lfloor n/2 \rfloor] \). This is because elements past \( \lfloor n/2 \rfloor \) have nothing to compare against, so by default \( A[\lfloor n/2 \rfloor + 1 \ldots n] \) satisfy the heap property (Fig. 7.5b).

After \( A[\lfloor n/2 \rfloor] \) is “in place,” everything from \( A[\lfloor n/2 \rfloor \ldots n] \) satisfies the heap property. We move back to the element at \( A[\lfloor n/2 \rfloor - 1] \) and sift it into place. This continues until we sift \( A[1] \) into place. At this point, \( A \) is a heap.

We use the heapify and sift functions to heapsort an array \( A \) as follows.

```plaintext
heapsort(A, n)

Input: \( A[] \), array to sort; \( n \), size of array

heapify( A, n ) // Convert A into a heap
i = n
while \( i \geq 2 \) do
    swap A[ 1 ], A[ i ] // Move largest heap element to sorted list
    i--
    sift( A, 1, i ) // Sift A[1] to satisfy heap constraints
end
```

Performance. Unlike Quicksort, heapsort has identical best case, worst case and
Chapter 7. Sorting

Figure 7.6. A heap represented as a binary tree, each node’s value is greater than or equal to the values in all of its child nodes.

average case performance\(^2,3\). Given this, we choose to explain a worst case scenario. We start with sift. The maximum number of swaps we could see would involve moving \(A[1]\) to \(A[2]\), then to \(A[4]\), and so on up to \(A[\lfloor n/2 \rfloor]\). Walking through \(\lfloor n/2 \rfloor\) elements in jumps of size 1, 2, 4 \ldots requires \(O(\lg n)\) steps.

The heapsort function first converts \(A\) into a heap, then applies sift \(n - 1\) times. The swap–sift loop therefore requires \(O(n \lg n)\) time to execute.

What about heapify? It calls sift \(\lfloor n/2 \rfloor\) times, so it needs no more than \(O(n \lg n)\) time to run. We could stop here, since this is equivalent to the time needed to sort the resulting heap. It turns out, however, that \(O(n \lg n)\) is not tight. heapify actually runs in \(O(n)\) time.

To understand this, you need to count the maximum number of jumps each element \(A[1], A[\lfloor n/2 \rfloor]\) would need to make to be pushed into place. Consider a heap drawn as a tree (Fig. 7.6). In this configuration, the heap property guarantees the value at any node is greater than or equal to the values of all of its children.

Given a heap forming a complete tree of height \(h\) with \(n = 2^{h+1} - 1\) elements, the \(2^h\) leaf elements are not pushed, the \(2^{h-1}\) elements at level \(h - 1\) can be pushed at most one time, the \(2^{h-2}\) elements at level \(h - 2\) can be pushed at most two times, and so on. In general, the \(2^{h-j}\) elements at level \(h - j\) can be pushed at most \(j\) times. The maximum number of sift pushes \(P(h)\) in a heap tree of height \(h\) is therefore

\[
P(h) = \sum_{j=1}^{h} j 2^{h-j} = \sum_{j=1}^{h} \frac{j 2^{h}}{2^j} = 2^h \sum_{j=1}^{h} \frac{j}{2^j} = 2^h \sum_{j=1}^{h} \frac{j}{2^j} = 2^h \sum_{j=1}^{h} \frac{j}{2^j} = 2^h \sum_{j=1}^{h} \frac{j}{2^j} (7.7)
\]

How do we solve \(\sum_{j=1}^{h} \frac{j}{2^j}\)? First, we can equate this to \(\sum_{j=0}^{h} \frac{j}{2^j}\), since \(\frac{j}{2^j} = 0\) for \(j = 0\). Next, consider the infinite geometric sum.

\[
\sum_{j=0}^{\infty} x^j = \frac{1}{1 - x} (7.8)
\]


If we differentiate both sides and multiply by $x$, we obtain

$$\sum_{j=0}^{\infty} (x^j)' x = \left(\frac{1}{1-x}\right)' x$$

$$\sum_{j=0}^{\infty} (jx^{j-1})' x = \left(\frac{1}{(1-x)^2}\right) x$$

$$\sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2}$$

(7.9)

If we replace $x$ with $\frac{1}{2}$, we obtain the function we want on the left hand side of the equation, $\sum_{j=0}^{\infty} j/2^j$. Substituting $x = \frac{1}{2}$ on the right hand side produces 2, so

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = 2$$

(7.10)

Our sum runs to $h$ and not $\infty$, so we’re bounded above by this equation.

$$P(n) = 2^h \sum_{j=1}^{\lfloor n/2 \rfloor} j/2^j$$

$$\leq 2^h \cdot 2 = 2^{h+1}$$

(7.11)

Since $n = 2^{h+1} - 1$, this simplifies to $P(n) \leq n + 1 = O(n)$. For completeness, since $\lfloor n/2 \rfloor$ elements must be considered for a push, $P(n)$ is also bounded below by $\Omega(n)$. Since $P(n)$ is bounded above by $O(n)$ and below by $\Omega(n)$, it has running time $\Theta(n)$.

### 7.4 Mergesort

Mergesort was one of the original sorting algorithms, proposed by John von Neumann in 1945. Similar to Quicksort, it works in a divide and conquer manner to sort an array $A$ of size $n$.

1. Divide $A$ into $\lfloor n/2 \rfloor$ sorted sublist, or runs, of length $l$.
2. Merge pairs of runs into new runs of size $2l$.
3. Continue merging runs until $l = n$, producing a sorted version of $A$.

In its most basic implementation we initially use $l \leq 1$, since a run of size zero or one is, by default, sorted. Runs are merged to produce new runs of size $2, 4 \ldots$ up to a final run of size $n$ (Fig. 7.7).

The bulk of the work is done in the merge step. This operation is simple: given two sorted runs, we walk through the runs in sequence, choosing the smaller of the
two values to output to a new, merged run. Assume the runs are contiguous in $A$, with the left run occupying $A[lf \ldots mid - 1]$ and the right run occupying $A[mid \ldots rt]$.

merge($A$, $lf$, $mid$, $rt$)

Input: $A[]$, runs to merge; $lf$, left run start; $mid$, right run start; $rt$, right run end

i = $lf$ // Current element in left run
j = $mid$ // Current element in right run
B = []
k = 0

while $k \leq rt - lf + 1$
do
  if $i < mid$ and ($j > rt$ or $A[i] \leq A[j]$)
    $B[k++] = A[i++]$ // Left run element exist, is smallest
  else
    $B[k++] = A[j++]$ // Right run element exists, is smallest
end

end

copy $B$ to $A[lf \ldots rt]$ // Copy merged runs

msort($A$, $beg$, $end$)

Input: $A[]$, array to split/merge; $beg$, start of subarray; $end$, end of subarray

if $end - beg \geq 1$
  mid = $\lfloor (beg + end) / 2 \rfloor$ // Start of right run
  msort($A$, $beg$, $mid - 1$) // Recursively create, sort left run
  msort($A$, $mid$, $end$) // Recursively create, sort right run
  merge($A$, $beg$, $mid$, $end$) // Merge sorted runs
end
The recursive function `msort` uses `merge` to split `A` into sorted runs, then merge the runs together to produce a sorted result. To sort `A`, we call `msort(A, 0, n-1)`.

**Performance.** In our implementation, best, average, and worst case performance for mergesort are all \( O(n \lg n) \).

First, consider the performance of `merge`. It walks through both runs, so it needs \( O(m) \) time, where \( m = rt - lf + 1 \) is the combined size of the runs.

Next, consider the recursive function `msort`. Similar to Quicksort, `msort` divides an array `A` of size \( n \) into two subarrays of size \( n/2 \), then four subarrays of size \( n/4 \), and so on down to \( n \) subarrays of size 1. The number of divisions needed is \( \lg n \), and the total amount of work performed to merge pairs of runs at each level of the recursion is \( O(n) \). Mergesort therefore runs in \( O(n \lg n) \) time.

Mergesort also requires \( O(n) \) additional space (the `B` array in the `merge` function) to hold merged runs.

## 7.5 Timsort

Timsort was proposed by Tim Peters in 2002. It was initially implemented as a standard sorting method in Python. It is now being offered as a built-in sorting method in environments like Android and Java. Timsort is a hybrid sorting algorithm, a combination of insertion sort and an adaptive mergesort, built specifically to work well on real-world data.

Timsort revolves around the idea that an array `A` is a sequence of sorted runs: ascending runs where \( A[i] \leq A[i+1] \leq A[i+2] \ldots \) and descending runs where \( A[i] > A[i+1] > A[i+2] \ldots \). Timsort leverages this fact by merging the runs together to sort `A`.

Every run will be at least 2 elements long\(^5\), but if `A` is random, very long runs are unlikely to exist. Timsort walks through the array, checking the length of each run it finds. If the run is too short, Timsort extends its length, then uses insertion sort to push the addition elements into sorted order.

How short is “too short?” Timsort defines a minimum run length `minrun`, based on the size of `A`.\(^6\) This guarantees that no run is less than `minrun` long, and for a random `A`, almost all the runs will be exactly `minrun` long, which leads to a very efficient mergesort.

As runs are identified or created, their starting position and length are stored on a run stack. Whenever a new run is added to the stack, a check is made to see whether any runs should be merged. Suppose that `X`, `Y`, and `Z` are the last three runs added to the top of the run stack. Only consecutive runs can be merged, so the two options are to create `(X + Y) Z` or `X (Y + Z)`.

Deciding when to merge is a balance between maintaining runs to possibly exploit good merges as new runs are found, versus merging quickly to exploit memory

---

\(^4\)Timsort reverses descending runs in-place, converting all runs to ascending.

\(^5\)A run starting with the last element in `A` will only be 1 element long.

\(^6\)`minrun` is selected from the range 32…65 if `n/minrun = 2^k` (i.e., a power of 2), or when this is not possible, `minrun ≈ 2^k` and `minrun < 2^k` (i.e., close to, but strictly less than a power of 2).
caching and to avoid a run stack that uses large amounts of memory. To do this, Timsort enforces two constraints on the lengths of the last three runs on the run stack.

1. \( X > Y + Z \)
2. \( Y > Z \)

If \( X \leq Y + Z \) the smaller of \( X \) and \( Z \) is merged with \( Y \), with ties favouring \( Z \). Fig. 7.8 shows two examples of merging \( Y \) with \( Z \) and \( X \) with \( Y \). Notice that in both cases the second constraint \( Y > Z \) is still violated, so we would continue merging the last three runs on the top of the stack until the constraints were satisfied, or until there is only one run on the stack.

**Merging.** Although it’s possible to apply a standard mergesort to merge two ascending runs \( X \) and \( Y \), Timsort tries to be smarter to improve absolute performance. Merging starts in the standard way, comparing \( X[0] \) to \( Y[0] \) and moving the smaller of the two to an output buffer. Timsort calls this *one pair at a time* mode. In addition to walking through the runs, we maintain a count \( c \) of how many times in a row the winning element comes from the same run.

**Galloping.** If \( c \) reaches a threshold *min-gallop*, we enter *galloping mode*. Now, we take element \( X[0] \) at the top of \( X \) and search \( Y \) directly for the position \( p \) where it belongs. We copy \( Y[0...p-1] \) to the output buffer, followed by \( X[0] \). Then we take \( Y[0] \) at the top of \( Y \) and search \( X \) for the position \( p \) where it belongs, copying \( X[0...p-1] \), then \( Y[0] \) to the output buffer. We continue galloping until both searches of \( X \) and \( Y \) copy subarrays that have less than *min-gallop* elements in them. At this point, we switch back to one pair at a time mode.

**Searching.** To find \( X[0] \) in \( Y \), we compare in turn to \( Y[0] \), \( Y[1] \), \( Y[3] \), \( Y[7] \), ... \( Y[2^k-1] \), searching for \( k \mid Y[2^k-1] < X[0] \leq Y[2^k-1] \). At this point we know that \( X[0] \) is somewhere in the \( 2^k-1 \) elements from \( Y[2^k-1] \) to \( Y[2^k-1] \). A regular binary search is used on this range to find the final position for \( X[0] \). The time needed to find \( k \) is \( \approx \lg |Y| \). Some extra time is also needed to perform a binary search on the
7.5. Timsort

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>identical</th>
<th>sequential</th>
<th>part seq</th>
<th>random</th>
<th>part rand</th>
</tr>
</thead>
<tbody>
<tr>
<td>mergesort</td>
<td>1.0×</td>
<td>1.0×</td>
<td>1.0×</td>
<td>1.0×</td>
<td>1.0×</td>
</tr>
<tr>
<td>Quicksort</td>
<td>6.52×</td>
<td>6.53×</td>
<td>1.81×</td>
<td>1.25×</td>
<td>1.53×</td>
</tr>
<tr>
<td>Timsort</td>
<td>6.87×</td>
<td>6.86×</td>
<td>1.28×</td>
<td>0.87×</td>
<td>1.6×</td>
</tr>
</tbody>
</table>

Table 7.2: Performance for optimized versions of mergesort, Quicksort, and Timsort on input data that is: identical, sequential, partially sequential, random without duplicates, and random with sequential steps, numbers indicate speedup versus mergesort

subarray containing X[0]’s position.

If we performed a binary search directly on Y to position X[0], it takes ⌈lg |Y| ⌉ comparisons, regardless of where X[0] lies in Y. This means that straight binary search only wins if the subarray identified using an adaptive search is large. It turns out that if the data in A is random, X[0] usually occurs near the front of Y, so long subarrays are extremely rare⁷. Even if long subarrays do occur, galloping still finds and copies them in O(lg n) versus O(n) for standard mergesort, producing a huge time savings.

Performance. In theoretical terms, Timsort’s best case performance is O(n), and its average and worst case performance are O(n lg n). However, since Timsort is tuned to certain kinds of real-world data—specifically, partially sorted data—it’s also useful to compare absolute performance for different types of input.

Table 7.2 shows absolute sort time speedups for Quicksort and Timsort versus mergesort for different input types⁸. Here, Timsort performed well for data that was sequential or partially random, which Quicksort performed best for data that was fully random. This suggests that Timsort has overall performance comparable to Quicksort, and if data is often sorted or nearly sorted, Timsort may outperform Quicksort.

