CHAPTER 3

Sorting

FIGURE 3.1 Frozen raspberries are sorted by size prior to packaging

SORTING IS one of the fundamental operations will we 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 theoretically cannot perform better than $O(n \lg n)$ in the best 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.

3.1 HEAPSORT

Heapsort is another common sorting algorithm. Unlike Quicksort, which runs in $O(n \lg n)$ in the best and average cases, but in $O(n^2)$ in the worse case, heapsort

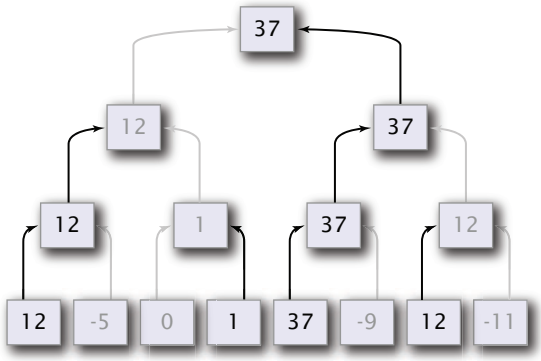

FIGURE 3.2 A tournament sort used to identify the largest number in the initial collection

guarantees $O(n \lg n)$ 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 (Figure 3.2). Once a winner is found, we reevaluate 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. Reevaluating a winning path requires $O(\lg n)$ time, since the height of the tournament tree is $\lg n$. Promoting all n values therefore requires $O(n \lg n)$ 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 begin, we define a heap as an array $A[1 \ldots n]$ that satisfies the following rule.[1]

$$\left.\begin{array}{l} A[i] \geq A[\,2i\,] \\ A[i] \geq A[\,2i+1\,] \end{array}\right\} \quad \text{if they exist} \tag{3.1}$$

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

[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.

1. Manipulate A into a heap.
2. Swap $A[1]$—the largest element in A—with $A[n]$, creating a heap with $n - 1$ elements and a partially sorted list with 1 element.
3. Readjust $A[1]$ as needed to ensure $A[1 \ldots n - 1]$ satisfy the heap property.
4. Swap $A[1]$—the second largest element in A—with $A[n - 1]$, creating a heap with $n - 2$ elements and a partially sorted list with 2 elements.
5. Continue readjusting 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: readjusting 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 (Figure 3.3b,c).

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

while i ≤ ⌊n/2⌋ do
    j = i*2                                                    // j = 2i
    k = j+1                                                  // k = 2i + 1

    if k ≤ n and A[k] ≥ A[j] then
        lg = k                              // A[k] exists and A[k] ≥ A[j]
    else
        lg = j                        // A[k] doesn't exist or A[j] > A[k]
    end

    if A[i] ≥ A[lg] then
        return                              // A[i] ≥ larger of A[j], A[k]
    end

    swap A[i], A[lg]
    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 = ⌊n/2⌋
while i ≥ 1 do
    sift( A, i, n )            // Sift A[i] to satisfy heap constraints
    i--
end
```

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 (Figure 3.3b).

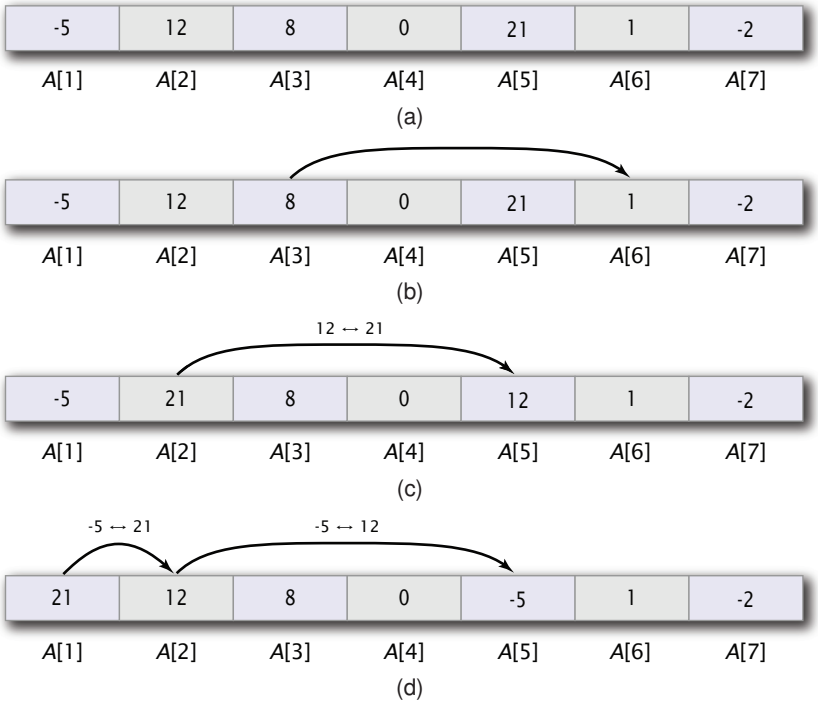

FIGURE 3.3 Heap construction: (a) A in its initial configuration; (b) sifting $A[3]$ to confirm its position; (c) sifting $A[2] = 12$ into place by swapping with $A[5] = 21$; (d) sifting $A[1] = -5$ into place by swapping with $A[2] = 21$, then with $A[5] = 12$

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.

```
heapsort(A, n)
Input: A[ ], array to sort; n, size of array
heapify( A, n )                                  // Convert A into a heap
i = n
while i ≥ 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

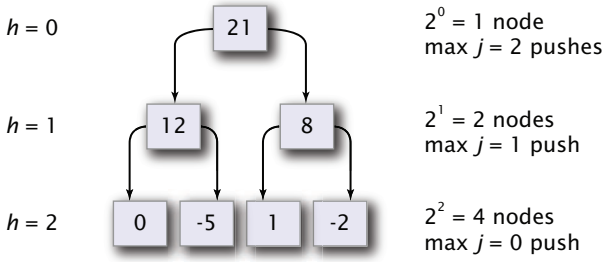

FIGURE 3.4 A heap represented as a binary tree; each node's value is greater than or equal to the values in all of its child nodes

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 ... 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 $A[1] \ldots A[\lfloor n/2 \rfloor]$ would need to make to be pushed into place. Consider a heap drawn as a tree (Figure 3.4). 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

$$P(h) = \sum_{j=1}^{h} j\,2^{h-j} = \sum_{j=1}^{h} j\frac{2^h}{2^j} = 2^h \sum_{j=1}^{h} \frac{j}{2^j} \tag{3.2}$$

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

$$\sum_{j=0}^{\infty} x^j = \frac{1}{(1-x)} \tag{3.3}$$

[2]The analysis of heapsort. Schaffer and Sedgewick. *Journal of Algorithms 15*, 1, 76–100, 1993.

[3]On the best case of heapsort. Bollobás, Fenner and Frieze. *Journal of Algorithms 20*, 2, 205–217, 1996.

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

$$\begin{aligned}\sum_{j=0}^{\infty}\left(x^j\right)' x &= \left(\frac{1}{(1-x)}\right)' x \\ \sum_{j=0}^{\infty}\left(jx^{j-1}\right) x &= \left(\frac{1}{(1-x)^2}\right) x \\ \sum_{j=0}^{\infty} jx^j &= \frac{x}{(1-x)^2}\end{aligned} \tag{3.4}$$

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

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = 2 \tag{3.5}$$

Our sum runs to h and not ∞, so we're bounded above by this equation.

$$\begin{aligned}P(n) &= 2^h \sum_{j=1}^{h} {}^{j}/_{2^j} \\ &\leq 2^h\, 2 = 2^{h+1}\end{aligned} \tag{3.6}$$

Since $n = 2^{h+1} - 1$, this simplifies to $P(n) \leq n + 1 = \mathrm{O}(n)$. 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 $\mathrm{O}(n)$ and below by $\Omega(n)$, it has running time $\Theta(n)$.

3.2 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 ${}^{n}/_{l}$ sorted sublists, 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 (Figure 3.5).

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

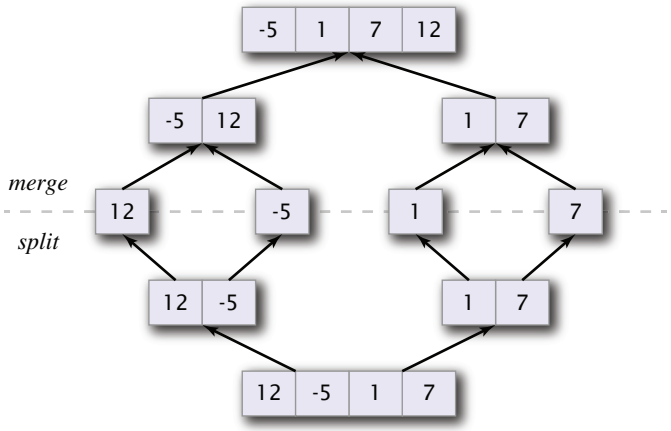

FIGURE 3.5 An array A split into runs of length 1, then recursively merged into a sorted result

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 ≤ rt - lf + 1 do
    if i < mid and (j > rt or A[i] ≤ A[j]) then
        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 ... 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 ≥ 1 then
    mid = ⌊( beg + end ) / 2⌋                        // 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.

3.3 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 $A[i] > A[i+1] > A[i+2] \ldots$.[4] 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 $32 \ldots 65 \mid n/minrun = 2^x$ (i.e., a power of 2), or when this is not possible, $\mid minrun \approx 2^x$ and $minrun < 2^x$ (i.e., close to, but strictly less than a power of 2).

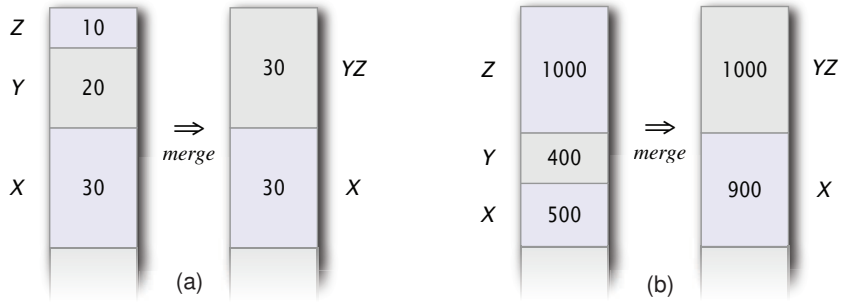

FIGURE 3.6 Managing the Timsort run stack: (a) $X \leq Y + Z$ and $Z < X$ so merge Y, Z; (b) $X \leq Y + Z$ and $X < Z$ so merge X, Y

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 favoring Z. Figure 3.6 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 $Y[0 \ldots p-1]$ 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 $X[0 \ldots p-1]$, 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 $Y[0]$, $Y[1]$, $Y[3]$, $Y[7]$, $\ldots$ $Y[2^j - 1]$, searching for $k \mid Y[2^{k-1} - 1] < X[0] \leq Y[2^k - 1]$. At this point we know that $X[0]$ is somewhere in the $2^{k-1} - 1$ elements from $Y[2^{k-1} - 1]$ 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 subarray containing $X[0]$'s position.

TABLE 3.1 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

Algorithm	identical	sequential	part seq	random	part rand
mergesort	1.0×	1.0×	1.0×	1.0×	1.0×
Quicksort	6.52×	6.53×	**1.81**×	**1.25**×	1.53×
Timsort	**6.87**×	**6.86**×	1.28×	0.87×	**1.6**×

If we perform a binary search directly on Y to position $X[0]$, it takes $\lceil \lg |Y| \rceil$ 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 rare.[7] 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 performances 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 3.1 shows absolute sort time speedups for Quicksort and Timsort versus mergesort for different input types.[8] Here, Timsort performed well for data that was sequential or partially random, while 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.

[7] Optimistic sorting and information theoretic complexity. McIlroy. *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1993),* Austin, TX, pp. 467–474, 1993.

[8] `http://blog.quibb.org/2009/10/sorting-algorithm-shootout`.