Sorting Algorithms

Overview

- Efficient comparison sorts
  - Merge Sort
  - Quicksort
- Non-comparison sorts
  - Counting Sort
  - Bucket Sort
- Hybrid sorting algorithms
  - Introsort
  - Timsort

Overview of sorting algorithms

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No
Insertion Sort	n	n^2	n^2	1	Yes
Merge Sort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	$O(n)$	Yes
Quicksort	$n\ log\ n$	n^2	$n\ log\ n$	n (worst case)	No*
Heapsort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	1	No
Counting Sort	$n + k$	$n + k$	$n + k$	$n + k$	Yes
Bucket Sort	$n + k$	n^2	$n + k$	$n \times k$	Yes
Timsort	n	$n\ log\ n$	$n\ log\ n$	n	Yes
Introsort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	$log\ n$	No

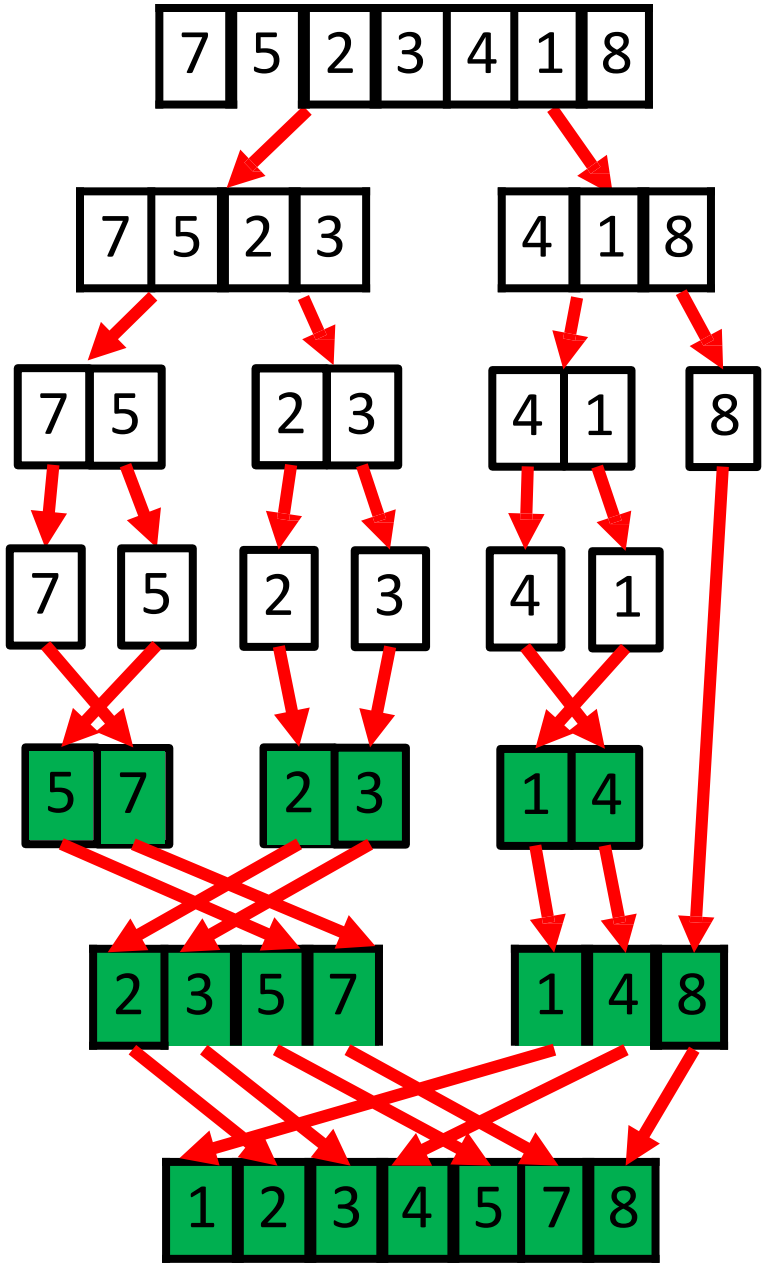
*the standard Quicksort algorithm is unstable, although stable variations do exist

Merge Sort

- John von Neumann -> 1945
- recursive divide-and-conquer approach - worst-case running time of $O(n \log n)$, the best asymptotic behaviour which we have seen so far.
- Its best, worst, and average cases are very similar, making it a very good choice if **predictable runtime** is important – Merge Sort gives good all-around performance.
- Stable sort
- Versions of Merge Sort are particularly good for sorting data with slow access times, such as data that cannot be held in internal memory (RAM) or are stored in linked lists.

Merge Sort example

- Mergesort is based on the following basic idea:
  - If the size of the list is 0 or 1, return.
  - Otherwise, separate the list into two lists of equal or nearly equal size and recursively sort the first and second halves separately.
  - Finally, merge the two sorted halves into one sorted list.
- Clearly, almost all the work is in the merge step, which should be as efficient as possible.
- Any merge must take at least time that is linear in the total size of the two lists in the worst case, since every element must be looked at in order to determine the correct ordering.



Quicksort

- C.A.R. Hoare -> 1959
- Like Merge Sort, Quicksort is a recursive Divide and Conquer algorithm
- Standard version is not stable, although stable versions do exist
- Performance: worst case n^2 (rare), average case $n \log n$, best case $n \log n$
- Memory usage: $O(n)$ (variants exist with $O(n \log n)$)
- In practice it is one of the fastest known sorting algorithms, on average

Quicksort procedure

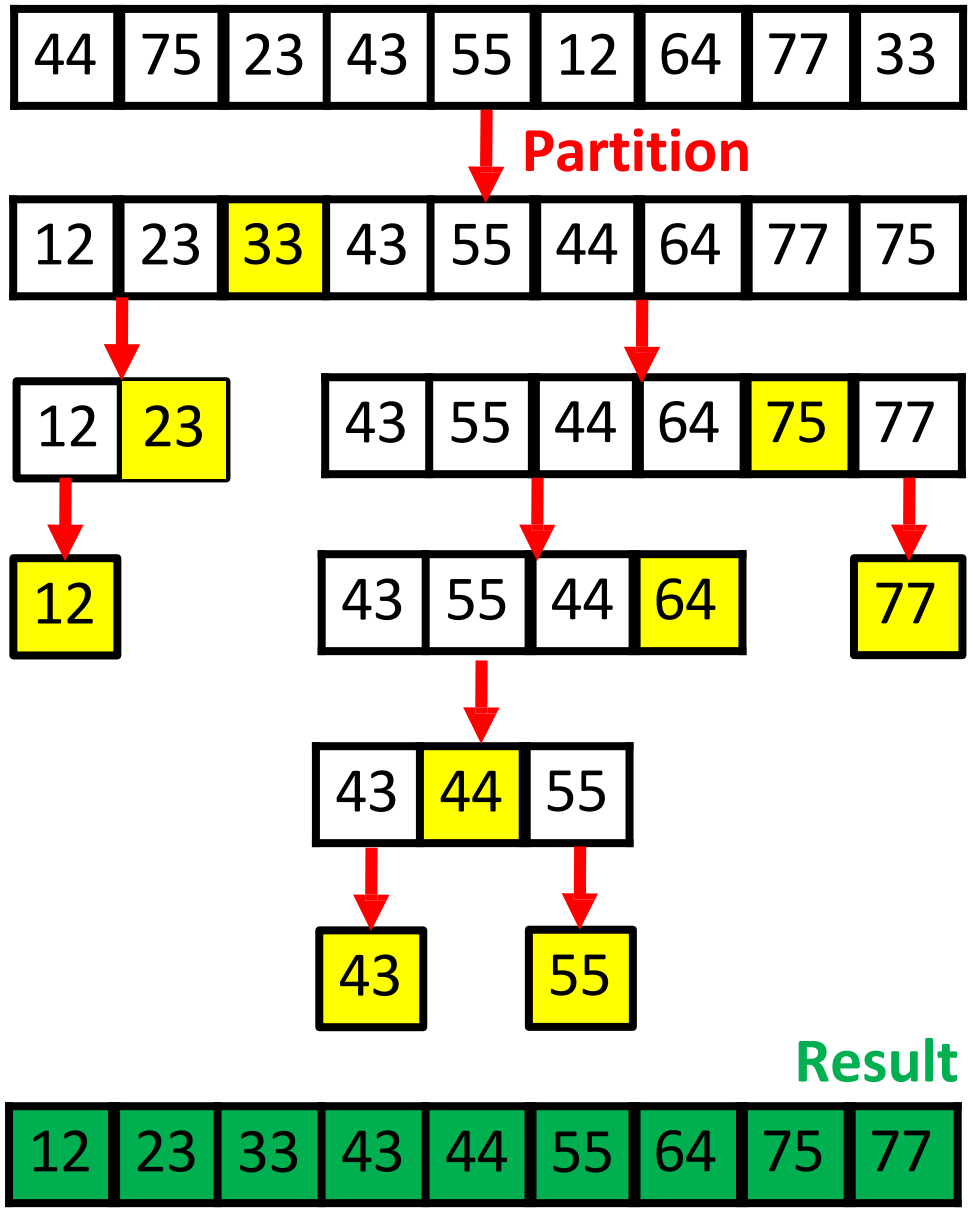
The main steps in Quicksort are:

1. Pivot selection: Pick an element, called a “pivot” from the array
2. Partitioning: reorder the array elements with values $<$ the pivot come before it, while all elements with values $\geq$ than the pivot come after it. After this partitioning, the pivot is in its final position.
3. Recursion: apply steps 1 and 2 above recursively to each of the two subarrays

The base case for the recursion is a subarray of length 1 or 0; by definition these cases do not need to be sorted

Quicksort example

- On average Quicksort runs in $n \log n$ but if it consistently chooses bad pivots, its performance degrades to n^2.
- This happens if the pivot is consistently chosen so that all (or too many of) the elements in the array are < the pivot or > than the pivot. (A classic case is when the first or last element is chosen as a pivot and the data is already sorted, or nearly sorted).
- Some options for choosing the pivot:
  - Always pick the first element as the pivot.
  - Always pick the last element as the pivot.
  - Pick a random element as the pivot.
  - Pick the median element as the pivot.



Non-comparison sorts

- “Comparison sorts” make no assumptions about the data and compare all elements against each other (the majority of sorting algorithms work in this way, including all sorting algorithms which we have discussed so far).
- $O(n \log n)$ time is the ideal “worst-case” scenario for a comparison-based sort (i.e. $O(n \log n)$ is the smallest penalty you can hope for in the worst case). Heapsort has this behaviour.
- $O(n)$ time is possible if we make assumptions about the data and don’t need to compare elements against each other (i.e., we know the data falls into a certain range or has some distribution).
- Examples of non-comparison sorts include Counting Sort, Bucket Sort and Radix Sort.
- $O(n)$ clearly is the minimum sorting time possible, since we must examine every element at least once (how can you sort an item you do not even examine?).

Counting Sort

- Proposed by Harold H. Seward in 1954.
- Counting Sort allows us to do something which seems impossible – sort a collection of items in (close to) linear time.
- How is this possible? Several assumptions must be made about the types of input instances which the algorithm will have to handle.
- i.e. assume an input of size n, where each item has a non-negative integer key, with a range of k (if using zero-indexing, the keys are in the range [0,…, k-1])
- Best-, worst- and average-case time complexity of $n + k$, space complexity is also $n + k$
- The potential running time advantage comes at the cost of having an algorithm which is not as widely applicable as comparison sorts.
- Counting Sort is stable (if implemented in the correct way!)

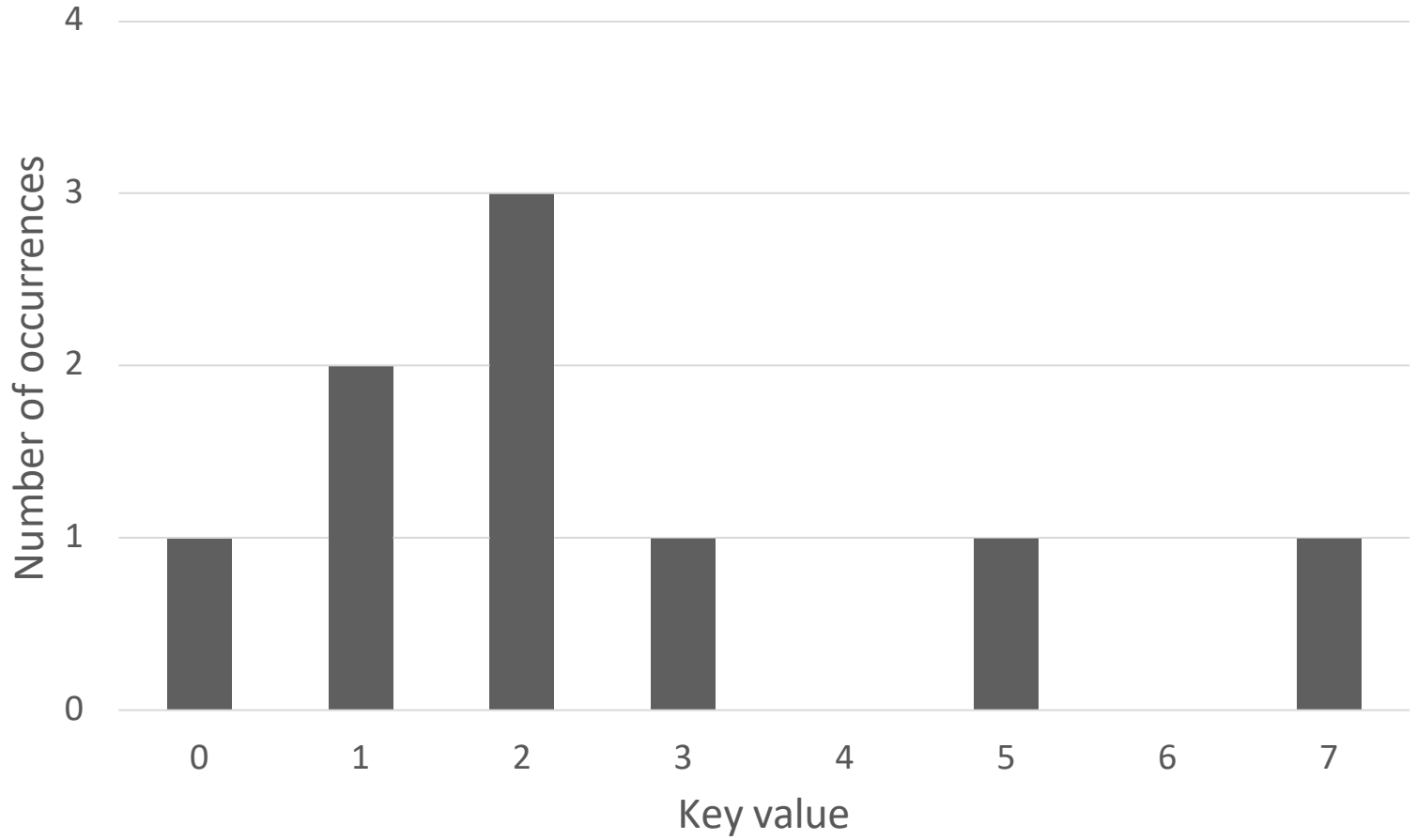
Counting Sort procedure

- Determine key range k of the `input` array (if not already known)
- Initialise an array `count` of size k, which will be used to count the number of times that each key value appears in the input instance.
- Initialise an array `result` of size n, which will be used to store the sorted output.
- Iterate through the input array, and record the number of times each distinct key value occurs in the input instance.
- Construct the sorted `result` array, based on the histogram of key frequencies stored in `count`. Refer to the ordering of keys in `input` to ensure that stability is preserved.

Counting Sort example

input

7
5
2
3
1
1
2
0
2



result

0
1
1
2
2
2
3
5
7

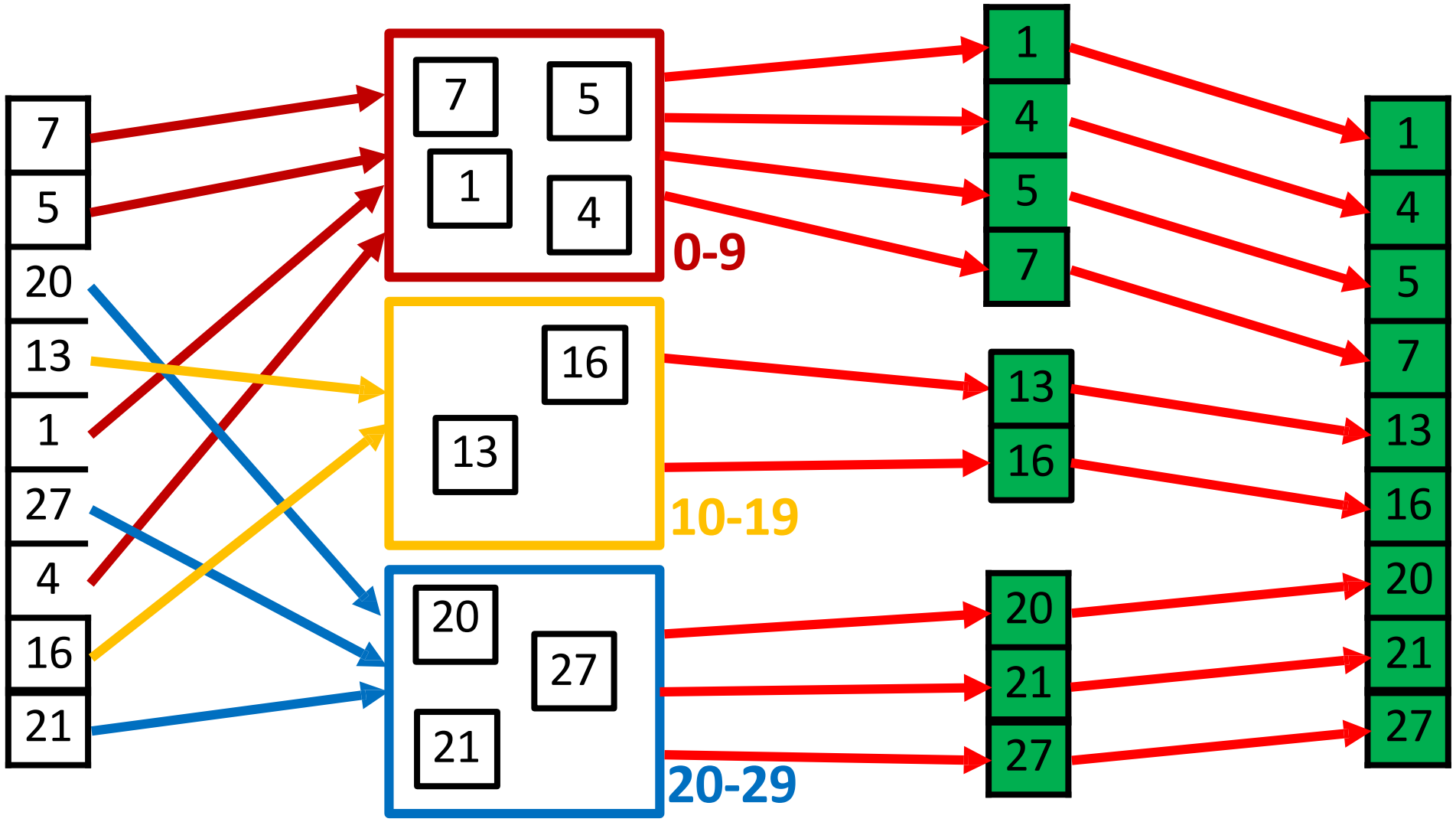
Bucket Sort

- Bucket Sort is a stable sort which works by distributing the elements of an array into a series of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the Bucket Sort algorithm
- Bucket Sort can be seen as a generalization of Counting Sort; in fact, if each bucket has size 1 then bucket sort degenerates to Counting Sort.
- Time complexity is n^2 in the worst case, and $n + k$ in the best and average cases (where k is the number of buckets)
- Worst case space complexity is $O(n \times k)$
- Bucket Sort is useful when input values are uniformly distributed over a range e.g. when sorting a large set of floating point numbers whose values are uniformly distributed between 0.0 and 1.0
- Bucket Sort's performance degrades with clustering; if many values occur close together, they will all fall into a single bucket and be sorted slowly

Bucket Sort procedure

- Set up an array of “buckets”, which are initially empty
- Iterate through the input array, placing each element into its correct bucket
- Sort each non-empty bucket (using either a recursive call to Bucket Sort, or a different sorting algorithm e.g. Insertion Sort)
- Visit the buckets in order, and place each element back into its correct position

Bucket Sort example



Hybrid sorting algorithms

- A hybrid algorithm is one which combines two or more algorithms which are designed to solve the same problem.
- Either chooses one specific algorithm depending on the data and execution conditions, or switches between different algorithms according to some rule set.
- Hybrid algorithms aim to combine the desired features of each constituent algorithm, to achieve a better algorithm in aggregate.
- E.g. The best versions of Quicksort perform better than either Heap Sort or Merge Sort on the vast majority of inputs. However, Quicksort has poor worst-case running time ($O(n^2)$) and $O(n)$ stack usage. By comparison, both Heap Sort and Merge Sort have $O(n \log n)$ worst-case running time, together with a stack usage of $O(1)$ for Heap Sort or $O(n)$for Merge Sort. Furthermore, Insertion Sort performs better than any of these algorithms on small data sets.

Introsort

- Hybrid sorting algorithm proposed by David Musser in 1997
- Variation of Quicksort which monitors the recursive depth of the standard Quicksort algorithm to ensure efficient processing
- If the depth of the Quicksort recursion exceeds $\log n$ levels, then Introsort switches to Heap Sort instead
- Since both algorithms which it uses are comparison-based, IntroSort is also comparison-based
- Fast average- and worst-case performance i.e. $n \log n$

Timsort

- Hybrid sorting algorithm implemented by Tim Peters in 2002 for use in the Python language.
- Derived from a combination of Merge Sort and Insertion Sort, along with additional logic (including binary search)
- Finds subsequences (runs) of the data that are already ordered, and uses that knowledge to sort the remainder more efficiently, by merging an identified run with existing runs until certain criteria are fulfilled.
- Used on the Android platform, Python (since 2.3), for arrays of non-primitive types in Java SE 7, and in the GNU Octave software

Criteria for choosing a sorting algorithm

Criteria	Sorting algorithm
Small number of items to be sorted	Insertion Sort
Items are mostly sorted already	Insertion Sort
Concerned about worst-case scenarios	Heap Sort
Interested in a good average-case behaviour	Quicksort
Items are drawn from a uniform dense universe	Bucket Sort
Desire to write as little code as possible	Insertion Sort
Stable sorting required	Merge Sort

Reference: Pollice G., Selkow S. and Heineman G. (2016). Algorithms in a Nutshell, 2nd Edition. O' Reilly.

Conclusion

- As we have seen, there are many different sorting algorithms, each of which has its own specific strengths and weaknesses
- Comparison-based sorts are the most widely applicable; but are limited to $n \log n$ running time in the best case
- Non-comparison sorts can achieve linear n running time in the best case, but are less flexible
- Hybrid sorting algorithms allow us to leverage the strengths of two or more algorithms (e.g. Timsort ≈ Merge Sort + Insertion Sort)
- There is no single algorithm which is best for all input instances; therefore it is important to use what you know about the expected input when choosing an algorithm