Sorting Algorithms

Part 1

Overview

- Introduction to sorting
- Conditions for sorting
- Comparator functions and comparison-based sorts
- Sort keys and satellite data
- Desirable properties for sorting algorithms
  - Stability
  - Efficiency
  - In-place sorting
- Overview of some well-known sorting algorithms
- Criteria for choosing a sorting algorithm

Sorting

- Sorting – arrange a collection of items according to some pre-defined ordering rules
- There are many interesting applications of sorting, and many different sorting algorithms, each with their own strengths and weaknesses.
- It has been claimed that as many as 25% of all CPU cycles are spent sorting, which provides a great incentive for further study and optimization
- The search for efficient sorting algorithms dominated the early days of computing.
- Numerous computations and tasks are simplified by properly sorting information in advance, e.g. searching for a particular item in a list, finding whether any duplicate items exist, finding the frequency of each distinct item, finding order statistics of a collection of data such as the maximum, minimum, median and quartiles.

Timeline of sorting algorithms

- 1945 – Merge Sort developed by John von Neumann
- 1954 – Radix Sort developed by Harold H. Seward
- 1954 – Counting Sort developed by Harold H. Seward
- 1959 – Shell Sort developed by Donald L. Shell
- 1962 – Quicksort developed by C. A. R. Hoare
- 1964 – Heapsort developed by J. W. J. Williams
- 1981 – Smoothsort published by Edsger Dijkstra
- 1997 – Introsort developed by David Musser
- 2002 – Timsort implemented by Tim Peters

Sorting

- Sorting is often an important step as part of other computer algorithms, e.g. in computer graphics (CG) objects are often layered on top of each other; a CG program may have to sort objects according to an “above” relation so that objects may be drawn from bottom to top
- Sorting is an important problem in its own right, not just as a pre-processing step for searching or some other task
- Real-world examples:
    - Entries in a phone book, sorted by area, then name
    - Transactions in a bank account statement, sorted by transaction number or date
    - Results from a web search engine, sorted by relevance to a query string

Conditions for sorting

- A collection of items is deemed to be “sorted” if each item in the collection is less than or equal to its successor
- To sort a collection A, the elements of A must be reorganised such that if $A[i] < A[j]$, then $i < j$
- If there are duplicate elements, these elements must be contiguous in the resulting ordered collection – i.e. if $A[i] = A[j]$ in a sorted collection, then there can be no k such that $i < k < j$ and $A[i] \neq A[k]$.
- The sorted collection A must be a permutation of the elements that originally formed A (i.e. the contents of the collection must be the same before and after sorting)

Comparing items in a collection

- What is the definition of "less than"? Depends on the items in the collection and the application in question
- When the items are numbers, the definition of "less than" is obvious (numerical ordering)
- If the items are characters or strings, we could use lexicographical ordering (i.e. apple < arrow < banana)
- Some other custom ordering scheme – e.g. Dutch National Flag Problem (Dijkstra), red < white < blue

Comparator functions

- Sorting collections of custom objects may require a custom ordering scheme
- In general: we could have some function compare(a,b) which returns:
  - -1 if a < b
  - 0 if a = b
  - 1 if a > b
- Sorting algorithms are independent of the definition of “less than” which is to be used
- Therefore we need not concern ourselves with the specific details of the comparator function used when designing sorting algorithms

Inversions

- The running time of some sorting algorithms (e.g. Insertion Sort) is strongly related to the number of ***inversions*** in the input instance.
- The number of ***inversions*** in a collection is one measure of how far it is from being sorted.
- An ***inversion*** in a list A is an ordered pair of positions (i, j) such that:
  - i < j but A[i] > A[j].
  - i.e. the elements at positions i and j are out of order
- E.g. the list [3,2,5] has only one inversion corresponding to the pair (3,2), the list [5,2,3] has two inversions, namely, (5,2) and (5,3), the list [3,2,5,1] has four inversions (3,2), (3,1), (2,1), and (5,1), etc.

Comparison sorts

- A comparison sort is a type of sorting algorithm which uses comparison operations only to determine which of two elements should appear first in a sorted list.
- A sorting algorithm is called ***comparison-based*** if the only way to gain information about the total order is by comparing a pair of elements at a time via the order ≤.
- Many well-known sorting algorithms (e.g. Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quicksort, Heapsort) fall into this category.
- Comparison-based sorts are the most widely applicable to diverse types of input data, therefore we will focus mainly on this class of sorting algorithms
- A fundamental result in algorithm analysis is that no algorithm that sorts by comparing elements can do better than $n \log n$ performance in the average or worst cases.
- Under some special conditions relating to the values to be sorted, it is possible to design other kinds of non-comparison sorting algorithms that have better worst-case times (e.g. Bucket Sort, Counting Sort, Radix Sort)

Sort keys and satellite data

- In addition to the **sort key** (the information which we use to make comparisons when sorting), the elements which we sort also normally have some **satellite data**
- Satellite data is all the information which is associated with the sort key, and should travel with it when the element is moved to a new position in the collection
- E.g. when organising books on a bookshelf by author, the author’s name is the sort key, and the book itself is the satellite data
- E.g. in a search engine, the sort key would be the relevance (score) of the web page to the query, and the satellite data would be the URL of the web page along with whatever other data is stored by the search engine
- For simplicity we will sort arrays of integers (sort keys only) in the examples, but note that the same principles apply when sorting any other type of data

Desirable properties for sorting algorithms

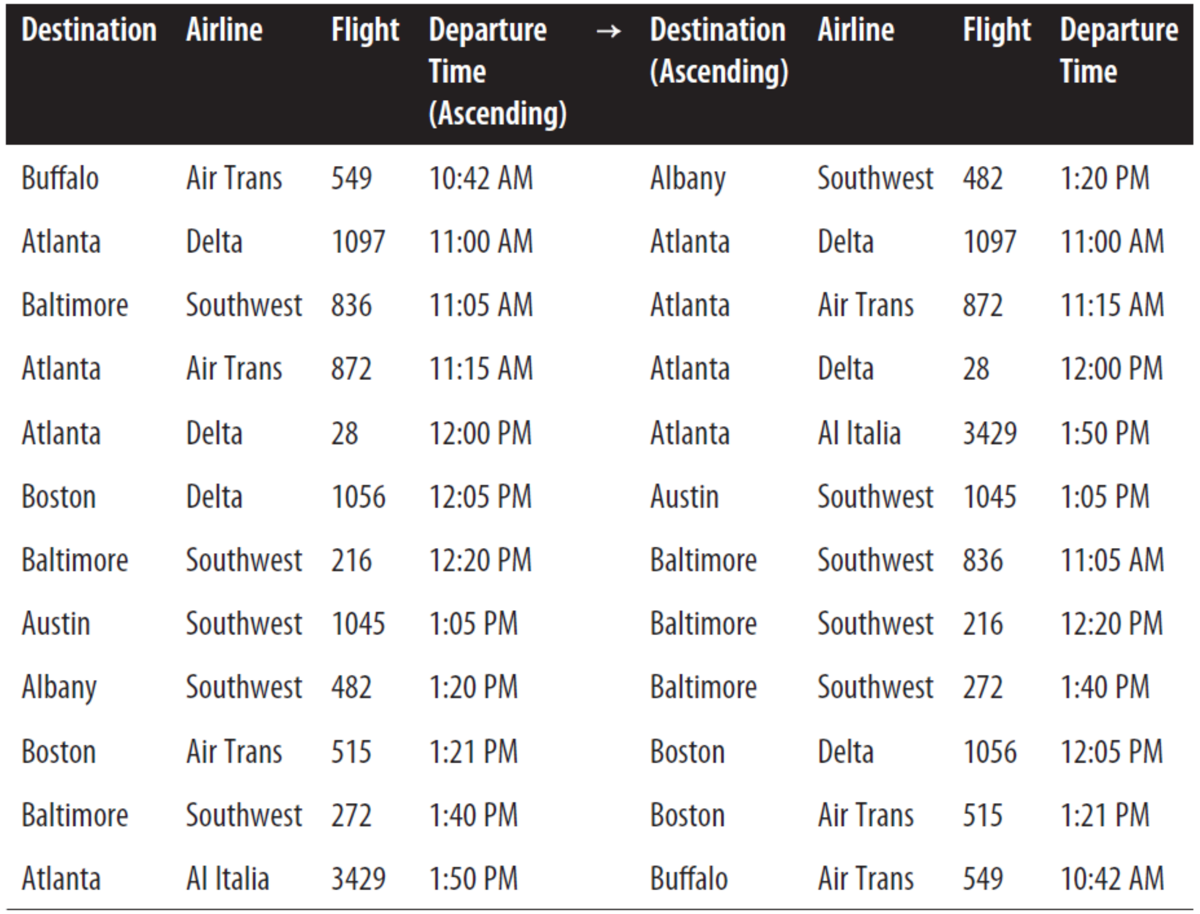
- Stability – preserve order of already sorted input
- Good run time efficiency (in the best, average or worst case)
- In-place sorting – if memory is a concern
- Suitability – the properties of the sorting algorithm are well-matched to the class of input instances which are expected i.e. consider specific strengths and weaknesses when choosing a sorting algorithm

Stability

- If a comparator function determines that two elements a_i and a_j in the original unordered collection are equal, it may be important to maintain their relative ordering in the sorted set
- i.e. if i < j, then the final location for A[i] must be to the left of the final location for A[j]
- Sorting algorithms that guarantee this property are **stable**
- **Unstable** sorting algorithms do not preserve this property
- Using an unstable sorting algorithm means that if you sort an already sorted array, the ordering of elements which are considered equal may be altered!

Stable sort of flight information

- All flights which have the same destination city are also sorted by their scheduled departure time; thus, the sort algorithm exhibited stability on this collection.
- An unstable algorithm pays no attention to the relationships between element locations in the original collection (it might maintain relative ordering, but it also might not).



Destination	Airline	Flight	Departure Time (Ascending)	→	Destination (Ascending)	Airline	Flight	Departure Time
Buffalo	Air Trans	549	10:42 AM		Albany	Southwest	482	1:20 PM
Atlanta	Delta	1097	11:00 AM		Atlanta	Delta	1097	11:00 AM
Baltimore	Southwest	836	11:05 AM		Atlanta	Air Trans	872	11:15 AM
Atlanta	Air Trans	872	11:15 AM		Atlanta	Delta	28	12:00 PM
Atlanta	Delta	28	12:00 PM		Atlanta	Al Italia	3429	1:50 PM
Boston	Delta	1056	12:05 PM		Austin	Southwest	1045	1:05 PM
Baltimore	Southwest	216	12:20 PM		Baltimore	Southwest	836	11:05 AM
Austin	Southwest	1045	1:05 PM		Baltimore	Southwest	216	12:20 PM
Albany	Southwest	482	1:20 PM		Baltimore	Southwest	272	1:40 PM
Boston	Air Trans	515	1:21 PM		Boston	Delta	1056	12:05 PM
Baltimore	Southwest	272	1:40 PM		Boston	Air Trans	515	1:21 PM
Atlanta	Al Italia	3429	1:50 PM		Buffalo	Air Trans	549	10:42 AM

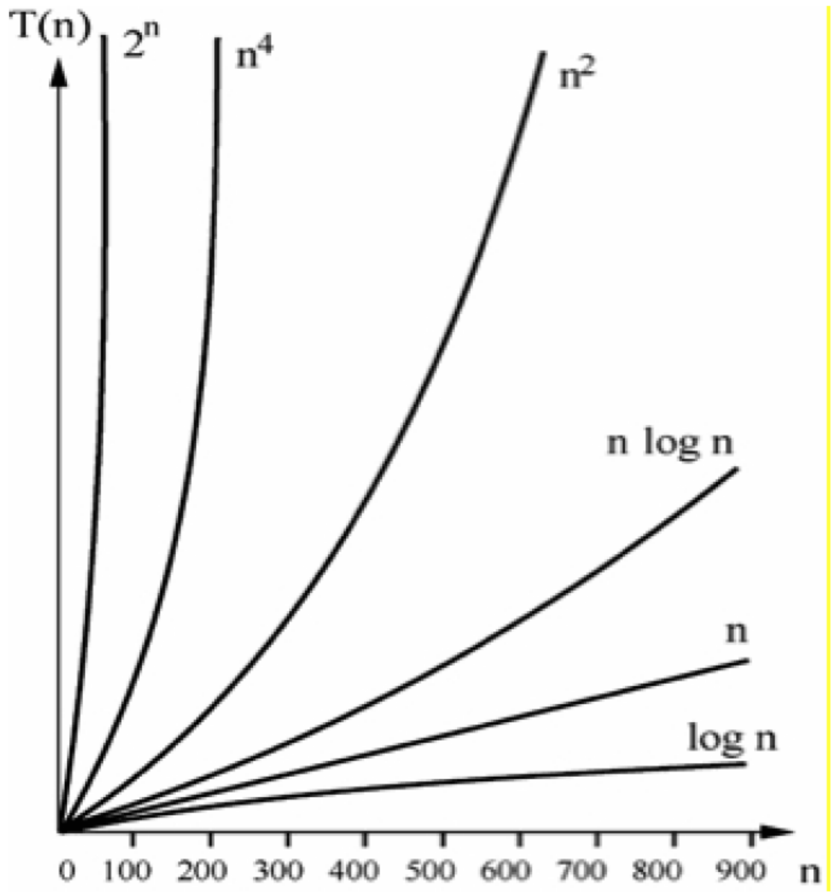
Reference: Pollice G., Selkow S. and Heineman G. (2016). Algorithms in a Nutshell, 2nd Edition. O' Reilly.

Analysing sorting algorithms

- When analysing a sorting algorithm, we must explain its best-case, worst-case, and average-case time complexity.
- The average case is typically hardest to accurately quantify and relies on advanced mathematical techniques and estimation. It also assumes a reasonable understanding of the likelihood that the input may be partially sorted.
- Even when an algorithm has been shown to have a desirable best-case, average-case or worst-case time complexity, its implementation may simply be impractical (e.g. Insertion Sort with large input instances).
- No one algorithm is the best for all possible situations, and so it is important to understand the strengths and weaknesses of several algorithms.

Recap: orders of growth

Running time $T(n)$ is proportional to:	Complexity:
$T(n) \propto \log n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n \log n$	linearithmic
$T(n) \propto n^2$	quadratic
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n;\ k > 1$	exponential



Factors which influence running time

- As well as the complexity of the particular sorting algorithm which is used, there are many other factors to consider which may have an effect on running time, e.g.
  - How many items need to be sorted
  - Are the items only related by the order relation, or do they have other restrictions (for example, are they all integers in the range 1 to 1000)
  - To what extent are the items pre-sorted
  - Can the items be placed into an internal (fast) computer memory or must they be sorted in external (slow) memory, such as on disk (so-called ***external sorting***).

In-place sorting

- Sorting algorithms have different memory requirements, which depend on how the specific algorithm works.
- A sorting algorithm is called ***in-place*** if it uses only a fixed additional amount of working space, independent of the input size.
- Other sorting algorithms may require additional working memory, the amount of which is often related to the size of the input n
- In-place sorting is a desirable property if the availability of memory is a concern

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

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.