Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 7.1.
Assistances and comments will be acknowledged.

7.1 Sorting

Sorting is the process of taking a list of *n* objects

$$a_0, a_1, \ldots, a_{n-1}$$

and rearranging these into a list

$$a'_0, a'_1, \ldots, a'_{n-1}$$

such that

$$a'_0 \le a'_1 \le \cdots \le a'_{n-1}.$$

This is essentially the process of converting an abstract list into an abstract sorted list.

In general, we will be deal with records that are associated with a number of fields. For example, the entries of Table 1 could be sorted numerically on the ID Number, as shown in Table 2, or sorted lexicographically based on surname first and then on the given name, as is shown in Table 3.

Table 1. A table of ID Nnumbers, names, and addresses.

19991532	Stevenson	Monica	3 Glenridge Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19985832	Khilji	Islam	37 Masterson Ave.
20003541	Groskurth	Ken	12 Marsdale Dr.
19981932	Carol	Ann	81 Oakridge Ave.
20019385	Beamer	Al	7 MacBeth Blvd.
19759325	Schellenberg	Al	153 Glen Morris Dr.
19982753	Beamer	Gord	17 Riverview Blvd.

Table 2. The records in Table 1 sorted by ID Number.

19759325	Schellenberg	Al	153 Glen Morris Dr.
19981932	Carol	Ann	81 Oakridge Ave.
19982753	Beamer	Gord	17 Riverview Blvd.
19985832	Khilji	Islam	37 Masterson Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glenridge Ave.
20003541	Groskurth	Ken	12 Marsdale Dr.
20019385	Beamer	Al	7 MacBeth Blvd.

Table 3. The records in Table 1 sorted by surname and then by given name.

20019385	Beamer	Al	7 MacBeth Blvd.
19982753	Beamer	Gord	17 Riverview Blvd.
19981932	Carol	Ann	81 Oakridge Ave.
20003541	Groskurth	Ken	12 Marsdale Dr.
19985832	Khilji	Islam	37 Masterson Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19759325	Schellenberg	Al	153 Glen Morris Dr.
19991532	Stevenson	Monica	3 Glenridge Ave.

For the purposes of this course, we will be sorting individual objects and not records.

We topic will conclude by considering some implementation details:

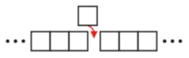
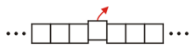
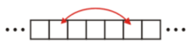
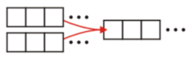
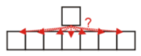
1. In-place sorting,
2. The classification of sorting algorithms,
3. The run-time of sorting algorithms, and
4. A lower bound on run-times,
5. A discussion on *optimal sorting* algorithms,
6. Two sub-optimal sorting algorithms (for fun), and
7. Inversions.

7.1.1 In-place Sorting

A sorting algorithm is said to be *in-place* if it uses Θ(1) memory—that is, it requires at most the local variables of the sorting function. Some of the sorting algorithms we will see will require a second array of equal size to the list being sorted; that is, Θ(n) additional memory. We will prefer in-place sorting algorithms.

7.1.2 Classification of Sorting Algorithms

The classification of sorting operations is based on the action performed:

1. Inserting a new object into a sorted list  ,
2. Selecting an item from an unsorted list ,
3. Exchanging two items in a list ,
4. Merging two sorted lists  , and
5. Distributing a value from a list  .

7.1.3 Run Times

We will be looking at seven algorithms that have run times of Θ(n), Θ(n ln(n)) and O(n^2). In each case, we will look at the average- and worst-case scenarios and in some cases, this will be distinctly different. Specifically, we will look at the algorithms in .

Table 4. Run-time of sorting algorithms.

Average Run Time	Algorithms	Comments
O(n^2)	insertion and bubble sort	
Θ(n ln(n))	heap, merge, and quick sort	The worst case for quick sort is Θ(n^2)
Θ(n)	bucket and radix sort	We must make assumptions about the data

7.1.4 Lower Bounds on Run Times

All sorting algorithms must inspect each object at least once requiring at least an $\Omega(n)$ run time. We will not be able to achieve this lower bound without additional assumptions.

In general, with any algorithm that uses comparisons, you can think of each comparison between two items as a branch in a binary tree: either the objects being compared are in order or they are out of order (inverted). You can think of all these comparisons as branches within a tree where the leaf nodes are all possible unsorted lists.

Now, there are $n!$ possible unsorted lists, and therefore, this would define a tree with $n!$ leaf nodes (the paths leading back to the root are the decisions that are made to sort the list) with a single sorted list at the root. A binary tree with n leaf nodes is of height at least $\lg(n)$ and therefore, the height of the tree will be at least $\lg(n!) = \Theta(n \ln(n))$.

A more detailed proof is given in Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.3.1, p.180.

7.1.5 A Comment on Optimality

There is no such thing as an *optimal* sorting algorithm. Many algorithms will be better under specific circumstances.

7.1.6 Sub-optimal Sorting Algorithms

There are two algorithms that have been studied for their run-time characteristics:

Bogo Sort

1. Check if the list is sorted, if it is, we are finished,
2. Otherwise, randomly reorder the entries of the list and go back to Step 1.

The best-case run time is $\Theta(n)$, the average-case run time is $\Theta(n\, n!)$, and in the worst case, it is unbounded.

Bozo Sort

1. Check if the list is sorted, if it is, we are finished,
2. Otherwise, randomly select two entries in the list, swap them, and go back to Step 1.

The run time of Bozo Sort appears to be marginally better than Bogo Sort on average: $\Theta(n!)$.

7.1.7 Inversions

How can we measure how *unsorted* a particular list is? For example, consider the following three lists:

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95
1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99
22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

A cursory inspection of these lists reveals that the first and second could be described as being *close* to being sorted while the third appears to be more random. Indeed, it takes only a few swaps to sort the first list. In the second list, even fewer operations are required, but 42 and 23, for example, is significantly out of place. The last list, however, would appear to be *unsorted* by any reasonable definition.

Now, given any list of n numbers, there are n-choose-2, or $\binom{n}{2} = \frac{n(n-1)}{2}$, pairs of numbers. For example, the unsorted list (1, 3, 5, 4, 2, 6) has the following 15 pairs

(1, 3)	(1, 5)	(1, 4)	(1, 2)	(1, 6)
	(3, 5)	(3, 4)	(3, 2)	(3, 6)
		(5, 4)	(5, 2)	(5, 6)
			(4, 2)	(4, 6)
				(2, 6)

Of these 15, 11 of the pairs appear in order:

(1, 3)	**(1, 5)**	**(1, 4)**	**(1, 2)**	**(1, 6)**
	(3, 5)	**(3, 4)**	(3, 2)	**(3, 6)**
		(5, 4)	(5, 2)	**(5, 6)**
			(4, 2)	**(4, 6)**
				(2, 6)

while the remaining four are out of order:

(1, 3)	(1, 5)	(1, 4)	(1, 2)	(1, 6)
	(3, 5)	(3, 4)	**(3, 2)**	(3, 6)
		(5, 4)	**(5, 2)**	(5, 6)
			(4, 2)	(4, 6)
				(2, 6)

We will describe these four pairs as being *inverted*. Thus, given any list (a_0, a_1, ..., a_{n-1}), we will define an inversion to be any pair (a_j, a_k) where $j < k$ but $a_j > a_k$. Thus, we will say that the list (1, 3, 5, 4, 2, 6) has four inversions.

One possible operation is to exchange (or swap) two adjacent entries in a list. You will note that any exchange of two adjacent entries in a list will either

1. Remove an inversion, or
2. Introduce a new inversion.

7.1.7.1 Number of Inversions

As there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs in any unsorted list, there are just as many possible inversions (the worst case occurring when the list is reverse sorted). Consequently, any algorithm that deals only with swapping adjacent pairs will have a run time up to O(n^2). By symmetry one could argue that we would expect in a randomly sorted list that there are approximately $\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4} = \Theta(n^2)$ inversions and therefore, the average run time of any algorithm that only swaps adjacent entries would also have an average run time of Θ(n^2).

If we consider the three previous lists, we note that each has 20 entries and therefore $\binom{20}{2} = \frac{20\cdot 19}{2} = 190$ pairs. In a randomly sorted list, we would expect approximately 95 inversions. With the first list,

1 **16 12** 26 25 35 33 58 **45 42** 56 67 83 75 74 **86 81** 88 99 95

there are 13 inversions:

(**16**, **12**) (26, 25) (35, 33) (58, 45) (58, 42) (58, 56) (**45**, **42**) (83, 75) (83, 74) (83, 81) (75, 74) (**86**, **81**) (99, 95).

In the second list,

1 17 21 **42** 24 27 32 35 45 47 57 **23** 66 69 70 76 **87 85** 95 99

there are also 13 inversions:

(**42**, 24) (**42**, 27) (**42**, 32) (**42**, 35) (**42**, **23**) (24, **23**) (27, **23**) (32, **23**) (35, **23**) (45, **23**) (47, **23**) (57, **23**) (**87**, **85**).

The third list, however, has 100 inversions.

7.1.7.2 Average Range of Inversions

Now, what is the probability that a randomly generated list will have close to $n(n-1)/4$ inversions? Unfortunately, as n becomes large, it becomes more-and-more likely. The standard deviation grows at a rate of $\sqrt{\tfrac{1}{72}n(n-1)(2n+5)} = \Theta(n^{3/2})$ which slower than the average, and therefore you are essentially guaranteed that a randomly generated list will have relatively close to the expected number of inversions. If you were to plot

$$\frac{n(n-1)}{4} \pm 1.96\cdot\sqrt{\frac{n(n-1)(2n+5)}{72}},$$

you would see the *95 % confidence interval* for any value of n. This is shown in Figure 1.

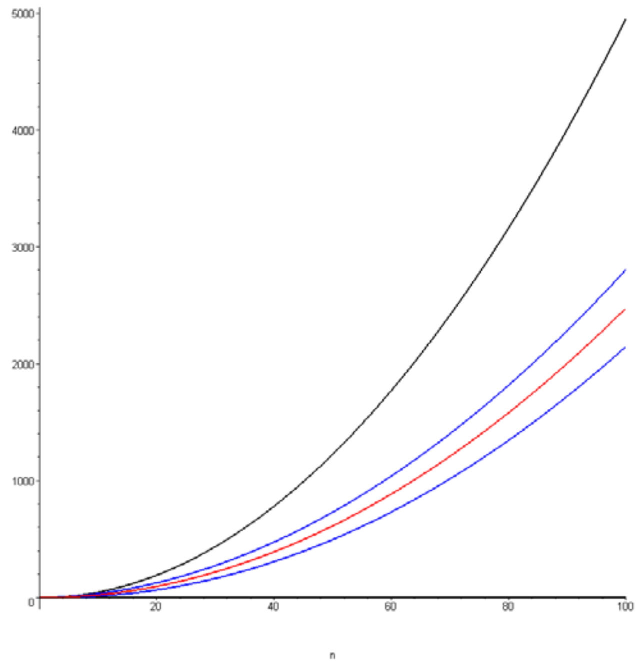


Figure 1. A 95 % confidence interval for the number of inversions in a randomly generated list.

A 95 % confidence interval says that 19 times out of 20, a result will appear in that interval. For example, in our previous example, with had a list size of $n = 20$ and therefore the confidence interval would be 95 ± 30.2 inversions. If increased the size of the array to 1000, we would expect 249 750 ± 10 338 inversions.