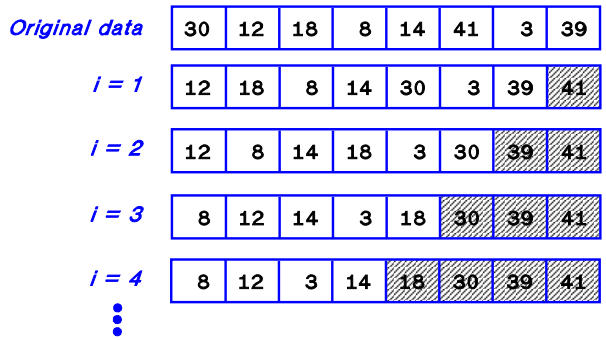
Bubble Sort

The bubble sort is the simplest but usually worst performing sort. Successive passes are made over the list – any time the left element of a pair is larger than the right, the two elements are swapped. The program terminates after n passes.



University of Idaho

Bubble Sort Improvements

1. Terminate after a pass that does no swaps.

➡ *List is already sorted.*

2. Do every other pass from end to beginning

➡ *0 to n passes move large elements quickly*

➡ *n downto 0 passes move small elements quickly*

University of Idaho

Bubble Sort Performance

Best Case: No swaps made on an already sorted array.

➡ *(n-1) comparisons, zero exchanges*

Worst Case: Sorted in reverse order

➡ *n(n-1)/2 comparisons, n(n-1)/2 exchanges*

Average case: Somewhat less than n-1 passes, only about half the exchanges.

➡ *n(n-x) comparisons, n(n-x)/4 exchanges*

Bubble sort is $O(n^2)$

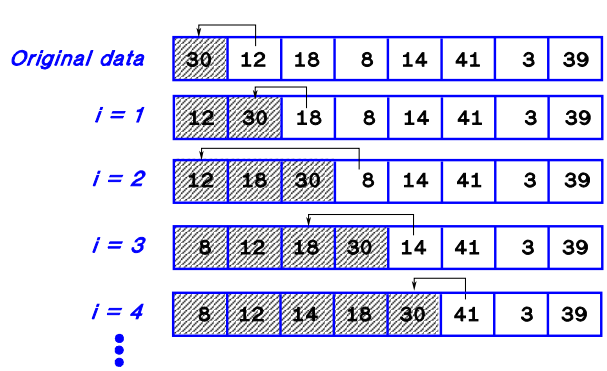
SORT0030

University of Idaho

Insertion Sort

Insertion sort works by "inserting" the next unsorted element into the sorted part of the list.

Initially, consider that the first element in the list is "sorted," and the rest of the elements are unsorted. Then insert, the second element, then the third, etc.



SORT0040

University of Idaho

Insertion Sort Performance/Improvements

Performance

Comparisons: For each pass we must search the sorted part of the list. There are n-1 passes. The list size changes each pass (1, 2, 3, ..., n-1), for an average of (n-1)/2. On average half the list will be searched each pass, so average number of comparisons is:

$$\frac{(n-1)(n-1)}{4} = O(n^2)$$

Moves: On average, we will need to move half the elements in the list each pass, leading to results similar to comparisons

Improvement

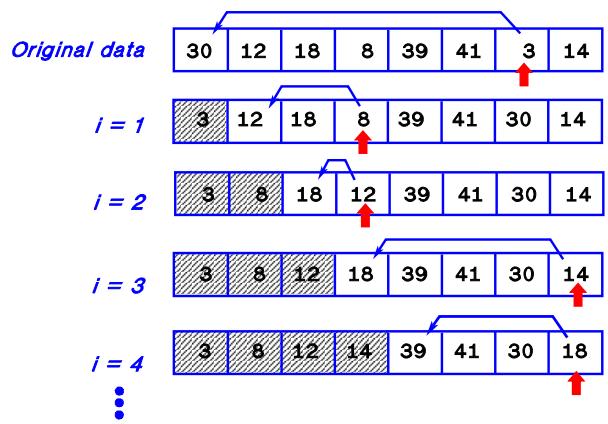
A search of the sorted part of the list is required. This could be replaced with a binary search, thereby reducing the number of comparisons to O(n log n) However, the number of moves remains the same.

SORT0050

University of Idaho

Selection Sort

Each pass of the selection sort finds the smallest remaining element in the unsorted list and adds it to the sorted part of the list. To "make room" for the new element in the list, the current value occupying the place is swapped with the small element.



SORT0060

University of Idaho

Selection Sort Performance

Comparisons: For each pass we must search the unsorted part of the list. There are n-1 passes, so the number of comparisions on average is:

$$\frac{(n-1)(n-1)}{4} = O(n^2)$$

Exchanges: Once the element has been selected, only one exchange (three moves) is required to put the element into place. Therefore, only one exchange per pass is performed, so exchanges are O(n).

The "Fast Sorts"

One class of sort algorithms rely on a technique called "Divide and Conquer" to achieve better performance than the previous sorts we studied.

Traditional sorts are of order $O(n^2)$. If we split the list in half and sort each half, the effort becomes:

$$O(\frac{n}{2})^2 + O(\frac{n}{2})^2 = O(\frac{n^2}{4})$$

While this is still $O(n^2)$, when applied successively and taken to the limit, the actual order can be reduced!

Divide and Conquer Algorithm

1. Apply an algorithm to break problem into two or more) parts (this is the divide step).

2. Apply an algorithm, either the original or one that performs the same function, to each of the two halves.

3. Apply an algorithm to recombine the problem.

If the divide and recombination algorithms are of order less than the original algorithm, the result will be an algorithm that is more efficient than the original!

The Merge Sort

1. Split the list in half.

   This is O(c)

2. Sort each half

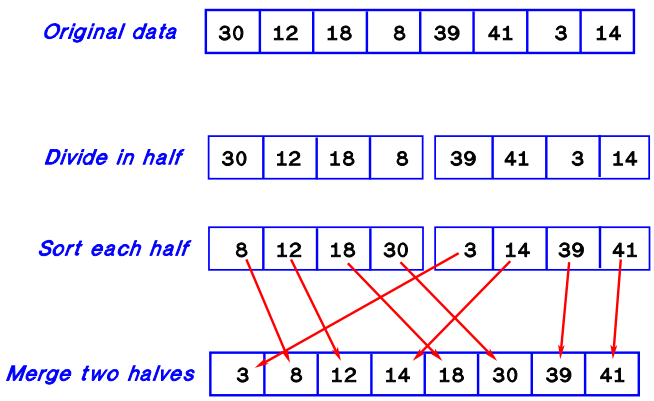
   Can use merge sort (recursion)

3. Merge the sorted halves into a single list

   This is O(n)

Merge Sort is O(n log n)!

Merge Sort Example



Quicksort

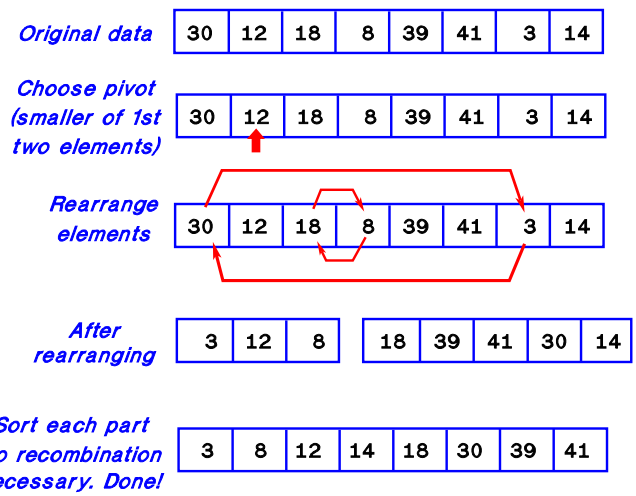
1. Split the list into two parts, around the "pivot" point. Arrange list so values to left of pivot are less than pivot, all values to right are greater than pivot. The two parts might not be equal in size.

   This is O(n)

2. Sort each part

   Can use any sort algorithm, including quicksort

3. (No recombination necessary – list is sorted!)

In general, Quicksort is O(n log n), and has best performance of all sorts! However, worst case is $O(n^2)$!

Quicksort Example