Part II

Algorithms

Chapter 8

Sorting

All the sorting algorithms in this chapter use data structures of a specific type to demonstrate sorting, e.g. a 32 bit integer is often used as its associated operations (e.g. $<$, $>$, etc) are clear in their behaviour.

The algorithms discussed can easily be translated into generic sorting algorithms within your respective language of choice.

8.1 Bubble Sort

One of the most simple forms of sorting is that of comparing each item with every other item in some list, however as the description may imply this form of sorting is not particularly effecient $O(n^2)$. In it's most simple form bubble sort can be implemented as two loops.

```
1) algorithm BubbleSort(list)
2)     Pre: list $\neq \emptyset$
3)     Post: list has been sorted into values of ascending order
4)     for $i \leftarrow 0$ to $list.Count - 1$
5)         for $j \leftarrow 0$ to $list.Count - 1$
6)             if $list[i] < list[j]$
7)                 $Swap(list[i], list[j])$
8)             end if
9)         end for
10)    end for
11)    return list
12) end BubbleSort
```

8.2 Merge Sort

Merge sort is an algorithm that has a fairly efficient space time complexity - $O(n\ log\ n)$ and is fairly trivial to implement. The algorithm is based on splitting a list, into two similar sized lists ($left$, and $right$) and sorting each list and then merging the sorted lists back together.

Note: the function MergeOrdered simply takes two ordered lists and makes them one.

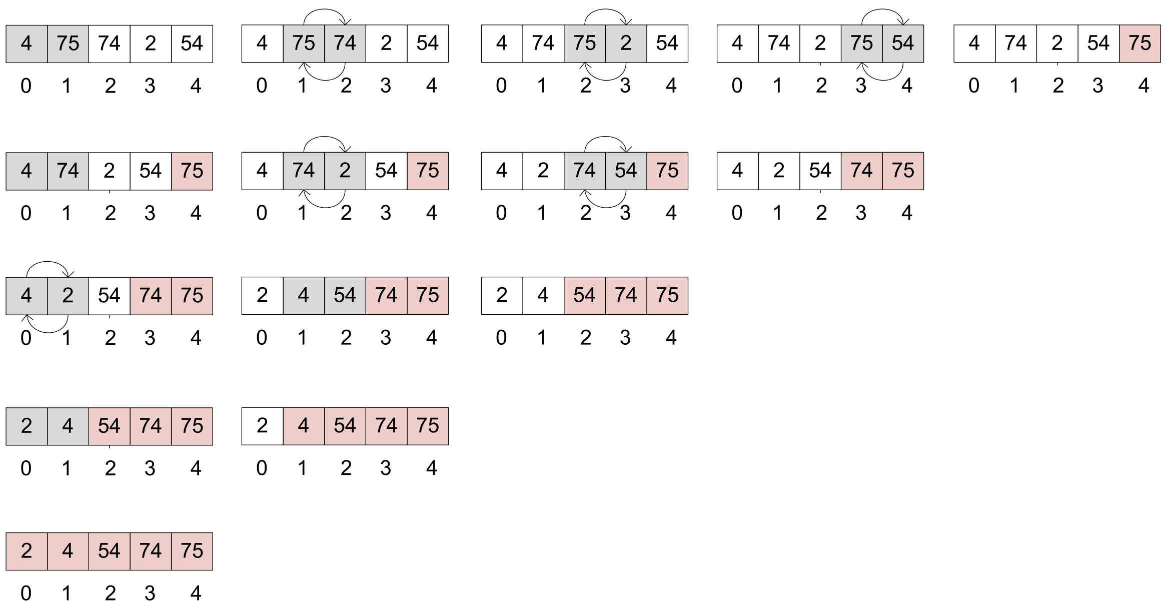

Figure 8.1: Bubble Sort Iterations

```
1) algorithm Mergesort(list)
2)     Pre: list ≠ ∅
3)     Post: list has been sorted into values of ascending order
4)     if list.Count = 1 // already sorted
5)        return list
6)     end if
7)     m ← list.Count / 2
8)     left ← list(m)
9)     right ← list(list.Count − m)
10)    for i ← 0 to left.Count−1
11)        left[i] ← list[i]
12)    end for
13)    for i ← 0 to right.Count−1
14)        right[i] ← list[i]
15)    end for
16)    left ← Mergesort(left)
17)    right ← Mergesort(right)
18)    return MergeOrdered(left, right)
19) end Mergesort
```

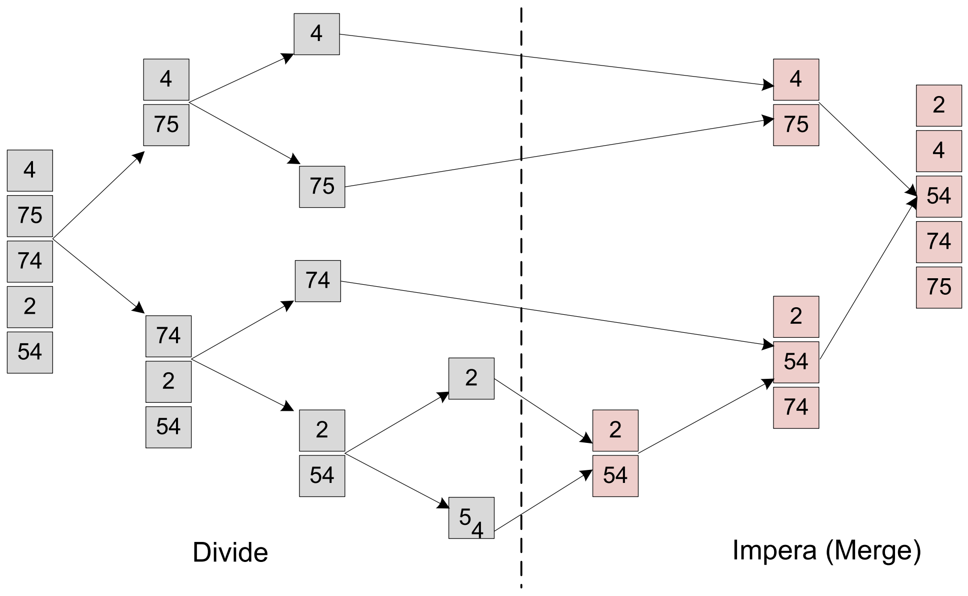

Figure 8.2: Merge Sort Divide et Impera Approach

8.3 Quick Sort

Quick sort is one of the most popular sorting algorithms based on divide et impera strategy, resulting in an $O(n\ log\ n)$ complexity. The algorithm starts by picking an item, called pivot, and moving all smaller items before it, while all greater elements after it. This is the main quick sort operation, called partition, recursively repeated on lesser and greater sub lists until their size is one or zero - in which case the list is implicitly sorted.

Choosing an appropriate pivot, as for example the median element is fundamental for avoiding the drastically reduced performance of $O(n^2)$.

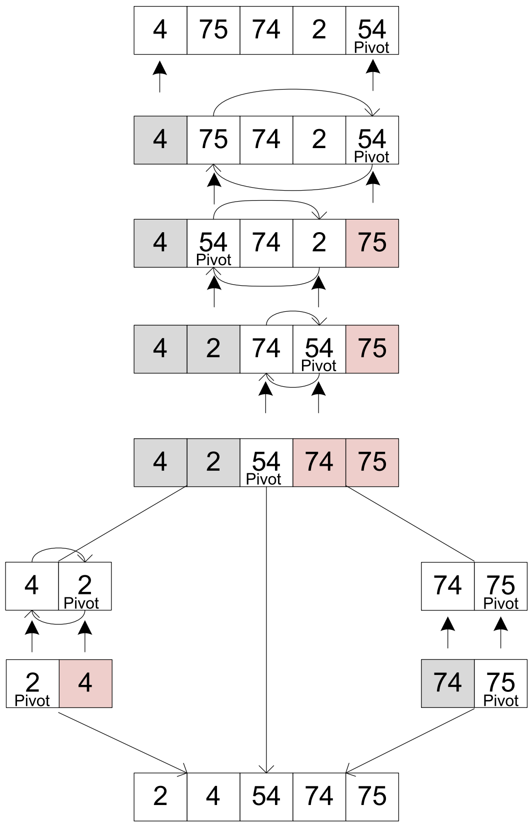

Figure 8.3: Quick Sort Example (pivot median strategy)

```
1) algorithm QuickSort($list$)
2)    Pre: $list \neq \emptyset$
3)    Post: $list$ has been sorted into values of ascending order
4)    if $list$.Count = 1 // already sorted
5)       return $list$
6)    end if
7)    $pivot \leftarrow$MedianValue($list$)
8)    for $i \leftarrow 0$ to $list$.Count$-1$
9)       if $list[i] = pivot$
10)         $equal$.Insert($list[i]$)
11)      end if
12)      if $list[i] < pivot$
13)         $less$.Insert($list[i]$)
14)      end if
15)      if $list[i] > pivot$
16)         $greater$.Insert($list[i]$)
17)      end if
18)   end for
19)   return Concatenate(QuickSort($less$), $equal$, QuickSort($greater$))
20) end Quicksort
```

8.4 Insertion Sort

Insertion sort is a somewhat interesting algorithm with an expensive runtime of $O(n^2)$. It can be best thought of as a sorting scheme similar to that of sorting a hand of playing cards, i.e. you take one card and then look at the rest with the intent of building up an ordered set of cards in your hand.

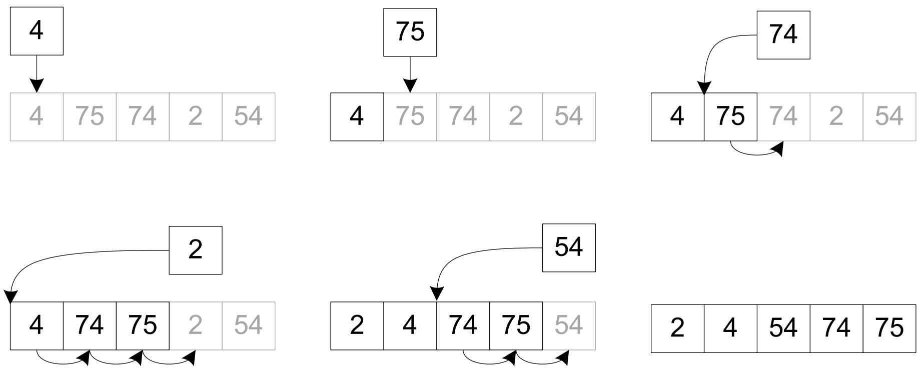

Figure 8.4: Insertion Sort Iterations

```
1) algorithm Insertionsort(list)
2)    Pre: $list \neq \emptyset$
3)    Post: list has been sorted into values of ascending order
4)    $unsorted \leftarrow 1$
5)    while $unsorted < list$.Count
6)       $hold \leftarrow list[unsorted]$
7)       $i \leftarrow unsorted - 1$
8)       while $i \geq 0$ and $hold < list[i]$
9)          $list[i + 1] \leftarrow list[i]$
10)         $i \leftarrow i - 1$
11)      end while
12)      $list[i + 1] \leftarrow hold$
13)      $unsorted \leftarrow unsorted + 1$
14)   end while
15)   return list
16) end Insertionsort
```

8.5 Shell Sort

Put simply shell sort can be thought of as a more efficient variation of insertion sort as described in §8.4, it achieves this mainly by comparing items of varying distances apart resulting in a run time complexity of $O(n\ log^2\ n)$.

Shell sort is fairly straight forward but may seem somewhat confusing at first as it differs from other sorting algorithms in the way it selects items to compare. Figure 8.5 shows shell sort being ran on an array of integers, the red coloured square is the current value we are holding.

```
1) algorithm ShellSort($list$)
2)     Pre: $list \neq \emptyset$
3)     Post: $list$ has been sorted into values of ascending order
4)     $increment \leftarrow list$.Count / 2
5)     while $increment \neq 0$
6)         $current \leftarrow increment$
7)         while $current < list$.Count
8)             $hold \leftarrow list[current]$
9)             $i \leftarrow current - increment$
10)            while $i \geq 0$ and $hold < list[i]$
11)                $list[i + increment] \leftarrow list[i]$
12)                $i -= increment$
13)            end while
14)            $list[i + increment] \leftarrow hold$
15)            $current \leftarrow current + 1$
16)        end while
17)        $increment\ /= 2$
18)    end while
19)    return $list$
20) end ShellSort
```

8.6 Radix Sort

Unlike the sorting algorithms described previously radix sort uses buckets to sort items, each bucket holds items with a particular property called a key. Normally a bucket is a queue, each time radix sort is performed these buckets are emptied starting the smallest key bucket to the largest. When looking at items within a list to sort we do so by isolating a specific key, e.g. in the example we are about to show we have a maximum of three keys for all items, that is the highest key we need to look at is hundreds. Because we are dealing with, in this example base 10 numbers we have at any one point 10 possible key values 0..9 each of which has their own bucket. Before we show you this first simple version of radix sort let us clarify what we mean by isolating keys. Given the number 102 if we look at the first key, the ones then we can see we have two of them, progressing to the next key - tens we can see that the number has zero of them, finally we can see that the number has a single hundred. The number used as an example has in total three keys:

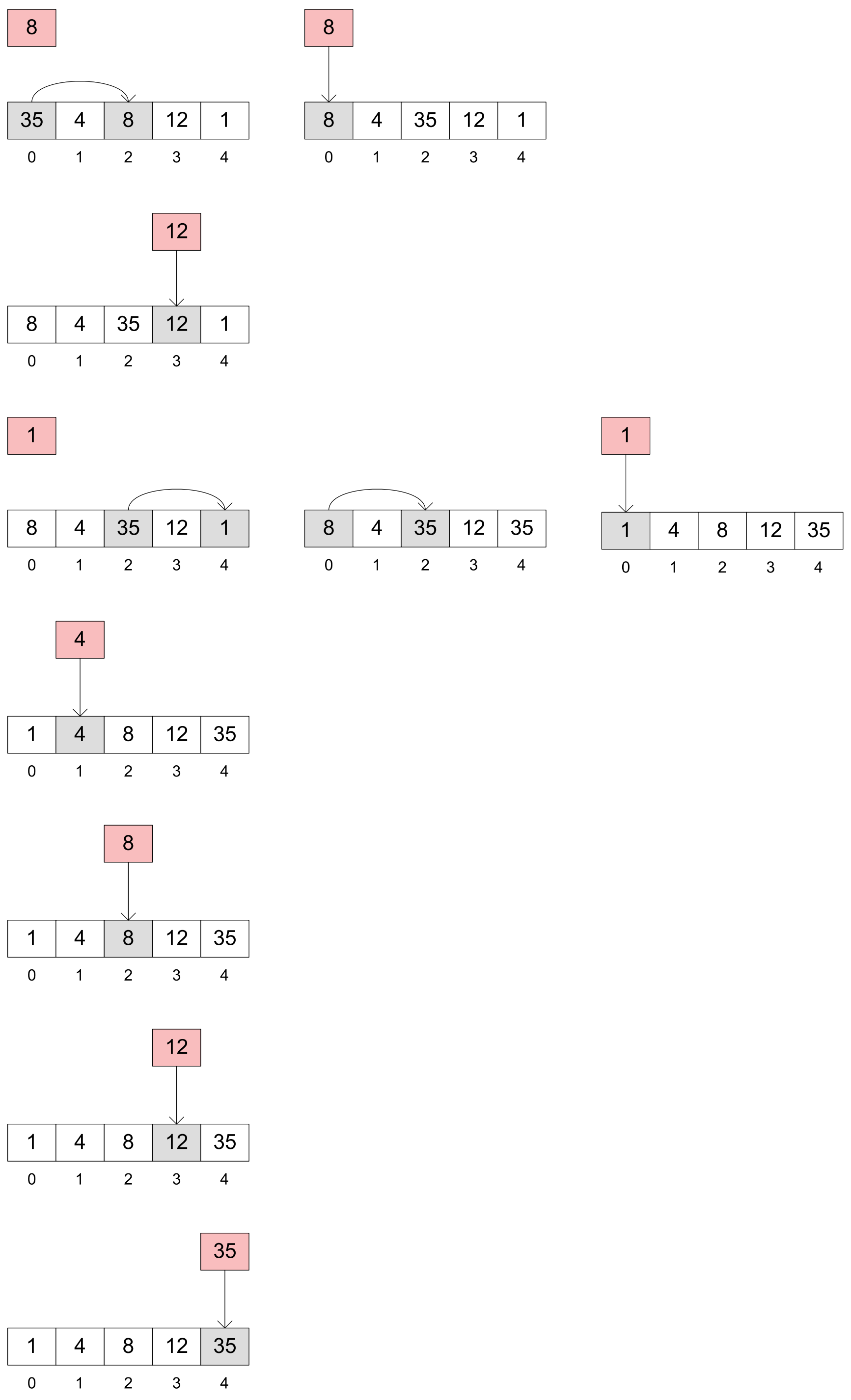

Figure 8.5: Shell sort

1. Ones
2. Tens
3. Hundreds

For further clarification what if we wanted to determine how many thousands the number 102 has? Clearly there are none, but often looking at a number as final like we often do it is not so obvious so when asked the question how many thousands does 102 have you should simply pad the number with a zero in that location, e.g. 0102 here it is more obvious that the key value at the thousands location is zero.

The last thing to identify before we actually show you a simple implementation of radix sort that works on only positive integers, and requires you to specify the maximum key size in the list is that we need a way to isolate a specific key at any one time. The solution is actually very simple, but its not often you want to isolate a key in a number so we will spell it out clearly here. A key can be accessed from any integer with the following expression: *key* ← (*number* / *keyToAccess*) % 10. As a simple example lets say that we want to access the tens key of the number 1290, the tens column is key 10 and so after substitution yields *key* ← (1290 / 10) % 10 = 9. The next key to look at for a number can be attained by multiplying the last key by ten working left to right in a sequential manner. The value of *key* is used in the following algorithm to work out the index of an array of queues to enqueue the item into.

```
algorithm Radix(list, maxKeySize)
  Pre: list $\neq \emptyset$
       maxKeySize $\geq 0$ and represents the largest key size in the list
  Post: list has been sorted
  queues ← Queue[10]
  indexOfKey ← 1
  fori ← 0 to maxKeySize − 1
    foreach item in list
      queues[GetQueueIndex(item, indexOfKey)].Enqueue(item)
    end foreach
    list ← CollapseQueues(queues)
    ClearQueues(queues)
    indexOfKey ← indexOfKey ∗ 10
  end for
  return list
end Radix
```

Figure 8.6 shows the members of *queues* from the algorithm described above operating on the list whose members are 90, 12, 8, 791, 123, and 61, the key we are interested in for each number is highlighted. Omitted queues in Figure 8.6 mean that they contain no items.

8.7 Summary

Throughout this chapter we have seen many different algorithms for sorting lists, some are very efficient (e.g. quick sort defined in §8.3), some are not (e.g.

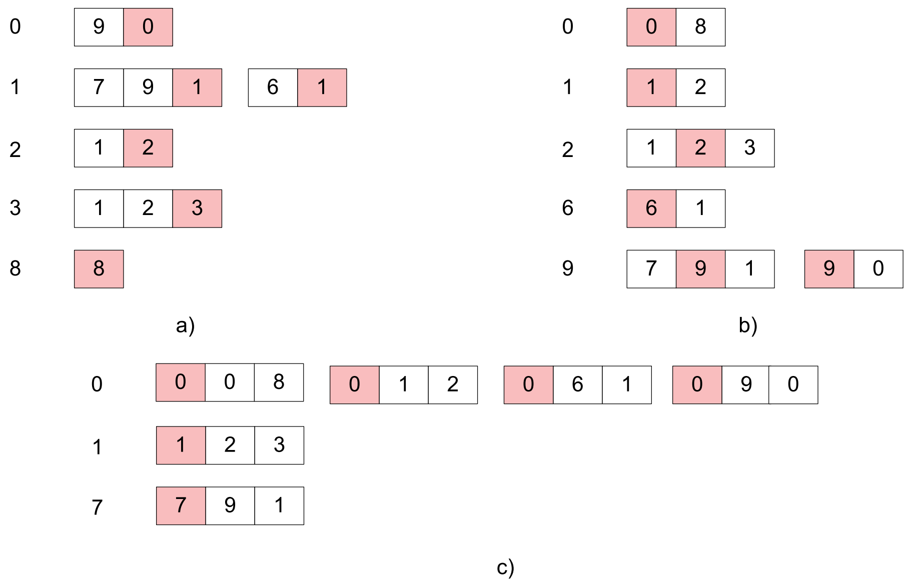

Figure 8.6: Radix sort base 10 algorithm

bubble sort defined in §8.1).

Selecting the correct sorting algorithm is usually denoted purely by efficiency, e.g. you would always choose merge sort over shell sort and so on. There are also other factors to look at though and these are based on the actual implementation. Some algorithms are very nicely expressed in a recursive fashion, however these algorithms ought to be pretty efficient, e.g. implementing a linear, quadratic, or slower algorithm using recursion would be a very bad idea.

If you want to learn more about why you should be very, very careful when implementing recursive algorithms see Appendix C.