Chapter 6: Sorting algorithms and complexity

Sorting is one of the most thoroughly studied algorithms in computer science. There are dozens of different sorting implementations, some applicable in general, others efficient in specific circumstances only.

Sorting can be used to solve a variety of problems, to mention a few basic ones:

- **Searching** for an item on a list works much faster if the list is sorted.
- **Selecting** items from a list based on their relationship to the rest of the items is easier with sorted data. For example, finding the k^{th}-largest or smallest value, or finding the median value of the list, is much easier when the values are in ascending or descending order.
- **Finding duplicate** values in a list can be done very quickly when the list is sorted.
- Analyzing the frequency **distribution** of items on a list is very fast if the list is sorted. For example, finding the element that appears most or least often is relatively straightforward with a sorted list.

Generate a list of random numbers to sort:

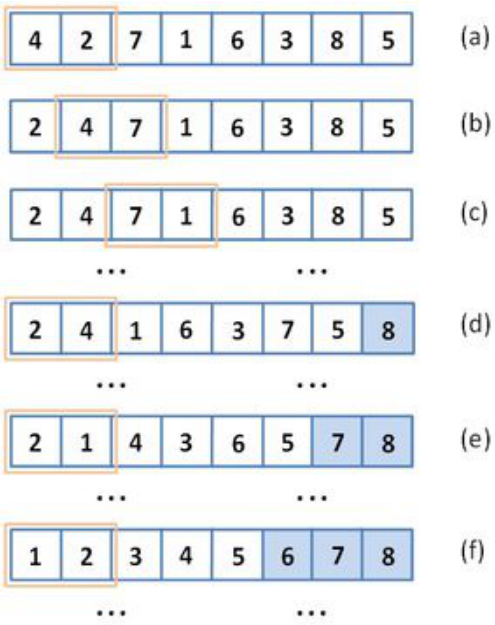
In [1]:

```
import random

originalNumbers = [random.randint(1, 100) for _ in range(20)]
print(originalNumbers)
```

```
[26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16, 52, 43, 7
1, 90, 17]
```

Bubble sort

Bubble sort is a simple sorting algorithms that works by repeatedly swapping the adjacent elements if they are in wrong order. In one iteration the largest element will be moved to the end of the array, thus reducting the problem to a shorter array.



In [2]:

```
def swap(array, i, j):
    temp = array[i]
    array[i] = array[j]
    array[j] = temp
```

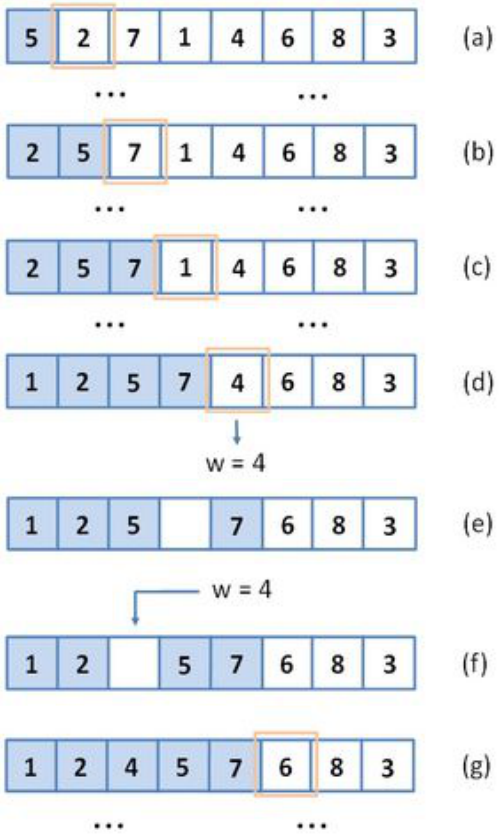
In [3]:

```
def bubbleSort(array):
    for end in range(len(array), 1, -1):
        for i in range(1, end):
            if array[i-1] > array[i]:
                swap(array, i-1, i)

numbers = originalNumbers.copy()
print("Unsorted: {0}".format(numbers))
bubbleSort(numbers)
print("Sorted: {0}".format(numbers))
```

```
Unsorted: [26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16,
52, 43, 71, 90, 17]
Sorted: [3, 4, 16, 17, 18, 21, 26, 32, 43, 51, 52, 52, 52, 62, 71, 7
4, 80, 88, 90, 98]
```

Insertion sort

Insertion sort is a simple sorting algorithm that maintains a sorted and an unsorted part of the array. Values from the unsorted part are picked and placed at the correct position in the sorted part.



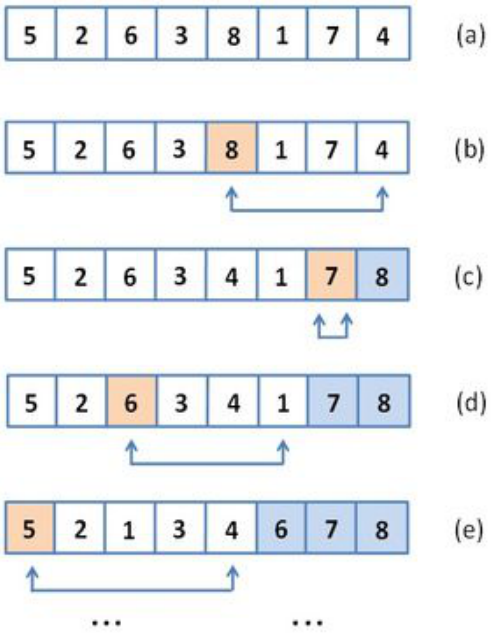
In [4]:

```
def insertionSort(array):
    for i in range(1, len(array)):
        value = array[i]
        j = i - 1
        while j >= 0 and array[j] > value:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = value

numbers = originalNumbers.copy()
print("Unsorted: {0}".format(numbers))
insertionSort(numbers)
print("Sorted: {0}".format(numbers))
```

```
Unsorted: [26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16,
52, 43, 71, 90, 17]
Sorted: [3, 4, 16, 17, 18, 21, 26, 32, 43, 51, 52, 52, 52, 62, 71, 7
4, 80, 88, 90, 98]
```

Maximum sort (a.k.a. Selection sort)

Maximum sort algorithm sorts an array of elements by repeatedly finding the maximum element (considering ascending order) from an unsorted part and putting it at the end of it. Then the length of the unsorted part is reduced by 1.
The algorithm can also formulated as a *Minimum sort* and combined they often they named *Selection sort*.



In [5]:

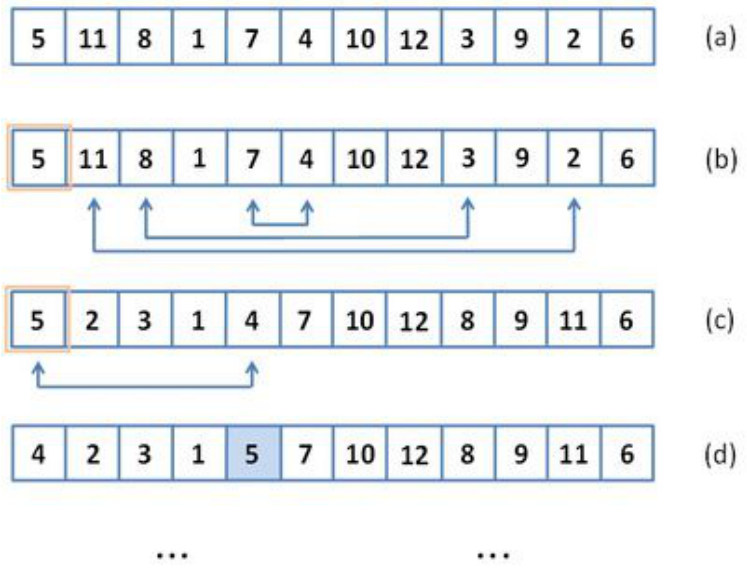
```
def maximumSort(array):
    for end in range(len(array), 1, -1):
        maxIdx = end - 1
        # maximum search algorithm
        for i in range(end):
            if array[i] > array[maxIdx]:
                maxIdx = i
        swap(array, end - 1, maxIdx)


numbers = originalNumbers.copy()
print("Unsorted: {0}".format(numbers))
maximumSort(numbers)
print("Sorted: {0}".format(numbers))
```

```
Unsorted: [26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16,
52, 43, 71, 90, 17]
Sorted: [3, 4, 16, 17, 18, 21, 26, 32, 43, 51, 52, 52, 52, 62, 71, 7
4, 80, 88, 90, 98]
```

Quicksort

Quicksort is a *Divide and Conquer* algorithm. It picks an element as *pivot* and partitions the given array around the picked pivot. The partitioning is executed that the algorithm puts the smaller element to the left of the pivot and the larger elements to the right of the pivot. Then the algorithm is executed recursively on the partitions.

There are many different versions on how to pick a "good" pivot element, the simplest solution is to always pick the first element.



Divide And Conquer algorithms in general works as follows:

- *Divide*: Divide the problem into more sub problems.
- *Conquer*: Solve the sub problems by calling recursively until sub problem solved is trivially solved.

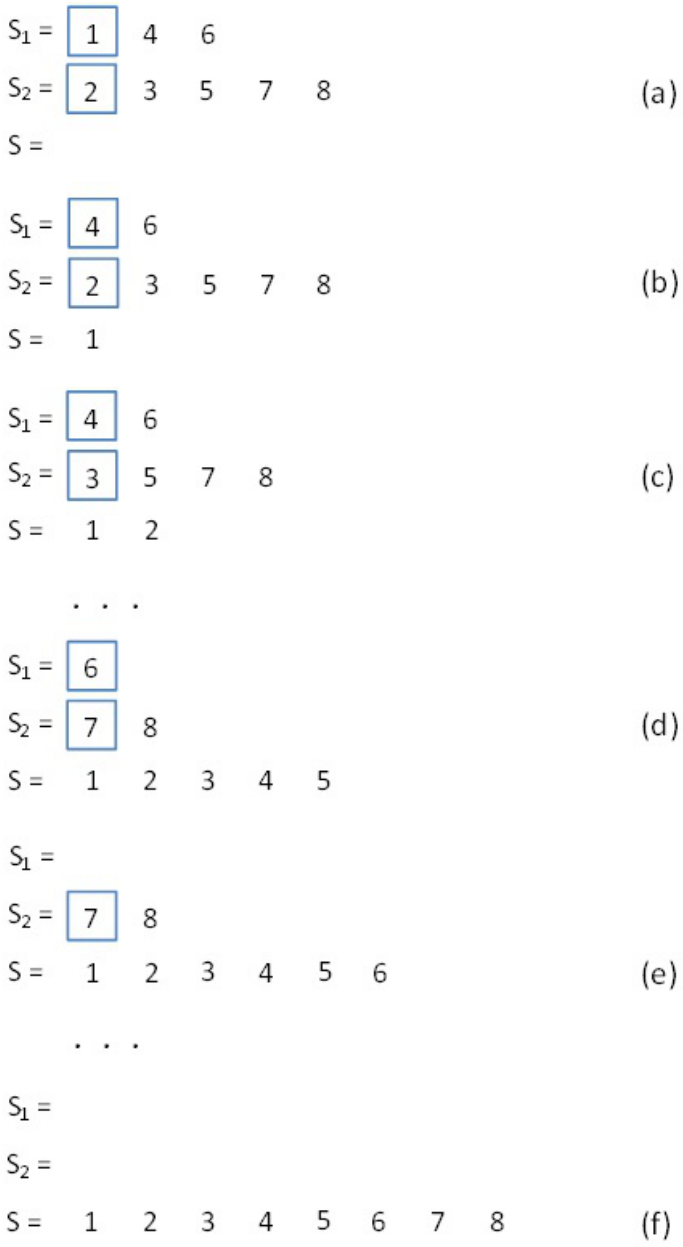
In [6]:

```
# Quicksorting
def quickSort(array):
    n = len(array)
    _quickSort(array, 0, n - 1)

# Quicksorting (partial array)
def _quickSort(array, u, v):
    if u >= v:
        return;

    k = _partition(array, u, v)
    _quickSort(array, u, k - 1)
    _quickSort(array, k + 1, v)

# Partinoning algorithm: move the pivot element to its position
def _partition(array, u, v):
    i = u + 1;
    j = v;
    while i <= j:
        while i <= v and array[i] <= array[u]:
            i += 1
        while j >= u + 1 and array[j] >= array[u]:
            j -= 1

        if i < j:
            swap(array, i , j)
            i += 1
            j -= 1

    swap(array, u, i - 1)
    return i - 1;


# Swap 2 items in a list
def swap(array, i, j):
    temp = array[i]
    array[i] = array[j]
    array[j] = temp

numbers = originalNumbers.copy()
print("Unsorted: %s" % numbers)
quickSort(numbers)
print("Sorted: %s" % numbers)
```

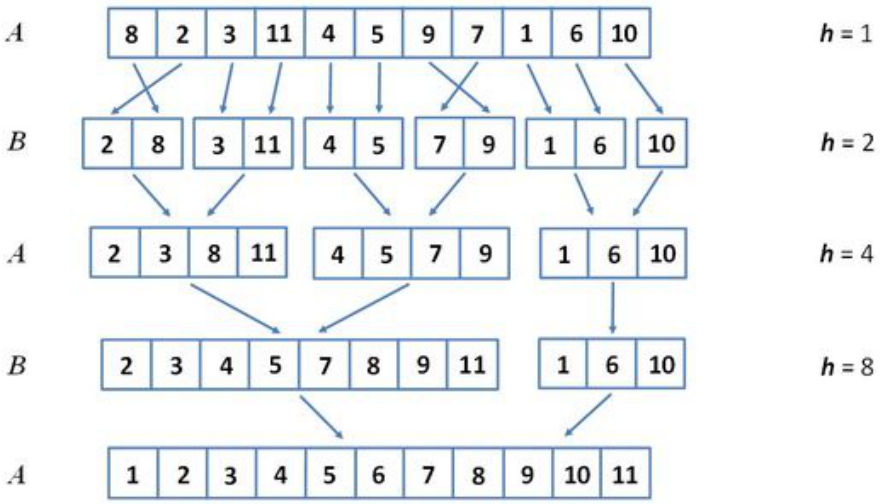
```
Unsorted: [26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16,
52, 43, 71, 90, 17]
Sorted: [3, 4, 16, 17, 18, 21, 26, 32, 43, 51, 52, 52, 52, 62, 71, 7
4, 80, 88, 90, 98]
```

Merge sort

Two sorted lists of data can be merged together by iterating through their elements only once.



The *Merge Sort* is also a *Divide and Conquer* algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. (We assume that the sorting of 2 elements is trivial.)



Remark: h denotes the (maximum) length of the sorted part of the array.

In [7]:

```
# Merge sort
def mergeSort(array):
    n = len(array)
    _mergeSort(array, 0, n - 1)

# Merge sort (partial array)
def _mergeSort(array, left, right):
    if left < right:
        middle = (left + right) // 2

        # Sort first and second halves
        _mergeSort(array, left, middle)
        _mergeSort(array, middle + 1, right)
        # Merge
        _merge(array, left, right, middle)

# Merges sorted partial arrays
def _merge(array, left, right, middle):
    nLeft = middle - left + 1
    nRight = right - middle

    # create temp arrays
    L = [0] * nLeft
    R = [0] * nRight

    # Copy data to temp arrays L[] and R[]
    for i in range(0 , nLeft):
        L[i] = array[left + i]

    for j in range(0 , nRight):
        R[j] = array[middle + 1 + j]

    # Initialize index positions
    i = 0
    j = 0
    k = left

    # Merge the temp arrays back into array[left..right]
    while i < nLeft and j < nRight:
        if L[i] <= R[j]:
            array[k] = L[i]
            i += 1
        else:
            array[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[]
    while i < nLeft:
        array[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[]
    while j < nRight:
        array[k] = R[j]
        j += 1
        k += 1
```

```
numbers = originalNumbers.copy()
print("Unsorted: %s" % numbers)
mergeSort(numbers)
print("Sorted: %s" % numbers)
```

```
Unsorted: [26, 4, 52, 74, 88, 51, 32, 21, 98, 3, 18, 52, 62, 80, 16,
52, 43, 71, 90, 17]
Sorted: [3, 4, 16, 17, 18, 21, 26, 32, 43, 51, 52, 52, 52, 62, 71, 7
4, 80, 88, 90, 98]
```

Complexity analysis of algorithms

In computer science, the analysis of algorithms is the process of finding the computational complexity of algorithms: the **amount of time, storage**, or other resources needed to execute them. Usually, this involves determining a function that relates the length of an algorithm's input to the number of steps it takes (its **time complexity**) or the number of storage locations it uses (its **space complexity**).

Now we will focus on **time complexity**. Let f and g represent the time complexity of 2 algorithms and we would like to make statements on how they grow compared to each other. (E.g. the time complexity of the *bubble sort* grows no faster than the n^2 function.)

In theoretical analysis of algorithms it is common to estimate their complexity in the *asymptotic sense*, i.e., to estimate the complexity function for arbitrarily large input. We are not concerned with small inputs or constant factors. The following notations are used to this end:

- **Big O (O) notation** describes the *asymptotic upper bound*, meaning that $f(N) = O(g(N))$, if such positive constants c and N_0 exists, that $f(N) \leq c * g(N)$, for all $N \geq N_0$.
- **Big-omega (Ω) notation** describes the *asymptotic lower bound*, meaning that $f(N) = O(g(N))$, if such positive constants c and N_0 exists, that $f(N) \geq c * g(N)$, for all $N \geq N_0$.
- **Big-theta (Θ) notation** describes the *asymptotic tight bound*, meaning that $f(N) = O(g(N))$, if such positive constants c_1, c_2 and N_0 exists, that $c_1 * g(N) \leq f(N) \leq c_2 * g(N)$, for all $N \geq N_0$.

In computer science in most cases we are interested in computing the *Big O* or the *Big-theta* notation, as a lower bound alone would not state much about the complexity.

For the most common complexities, well-known names have also be assigned and used:

Asympthotic complexity	Name
$\Theta(1)$	Constant time
$\Theta(n)$	Linear time
$\Theta(n^2)$	Quadratic time
$\Theta(n^3)$	Cubic time
$\Theta(log(n))$	Logarithmic time
$\Theta(n * log(n))$	Linearithmic time
$\Theta(2^n)$	Exponential time
$\Theta(n!)$	Factorial time

Question: what is the asymptotic computational complexity of the introduced sorting algoirhtms?

Bubbke sort, insertsion sort, maximum sort: $\Theta(n^2))$
Quicksort, merge sort: $\Theta(n * log(n))$

It can be proven that for a general case there is no better time complexity for sorting than $\Theta(n * log(n))$.
There are further algorithms with this complexity, see e.g. **Heap sort** (https://en.wikipedia.org/wiki/Heapsort) or **Tournament sort** (https://en.wikipedia.org/wiki/Tournament_sort).