In [12]:

```
#Selection sort (asc)
def selection_sort_min(list1):
    for idx in range(len(list1)):
        minimum = list1[idx]
        min_idx = idx
        for j in range(idx+1, len(list1)):
            if list1[j]<minimum:
                minimum = list1[j]
                min_idx = j
        #swap list1[min_idx] with list1[idx]
        list1[min_idx], list1[idx] = list1[idx], list1[min_idx]
        return list1

#Selection sort (desc)
def selection_sort_max(list1):
    for idx in range(len(list1)):
        maximum = list1[idx]
        max_idx = idx
        for j in range(idx+1, len(list1)):
            if list1[j]>maximum:
                maximum = list1[j]
                max_idx = j
        #swap list1[min_idx] with list1[idx]
        #list1[max_idx], list1[idx] = list1[idx], list1[max_idx]
        temp = list1[max_idx]
        list1[max_idx] = list1[idx]
        #list1[idx] = list1[max_idx]
        list1[idx] = temp
        return list1


numbers = [12,-3,40,133,300,32,34,1,7]
print(numbers)
print("==================================")
sorted_numbers_min = selection_sort_min(numbers)
print(sorted_numbers_min)
sorted_numbers_max = selection_sort_max(numbers)
print(sorted_numbers_max)
```

```
[12, -3, 40, 133, 300, 32, 34, 1, 7]
==================================
```

```
[-3, 12, 40, 133, 300, 32, 34, 1, 7]
[300, 12, 40, 133, -3, 32, 34, 1, 7]
```

In [9]:

```
#Selection sort for both ascending and descending using a single function

def selection_sort(list1, reverse = False):
    if reverse == False:

        for idx in range(len(list1)):
            minimum = list1[idx]
            min_idx = idx
            for j in range(idx+1, len(list1)):
                if list1[j]<minimum:
                    minimum = list1[j]
                    min_idx = j
            #swap list1[min_idx] with list1[idx]
            list1[min_idx], list1[idx] = list1[idx], list1[min_idx]
        return list1

    else:
        for idx in range(len(list1)):
            maximum = list1[idx]
            max_idx = idx
            for j in range(idx+1, len(list1)):
                if list1[j]>maximum:
                    maximum = list1[j]
                    max_idx = j
            #swap list1[min_idx] with list1[idx]
            #list1[max_idx], list1[idx] = list1[idx], list1[max_idx]
            temp = list1[max_idx]
            list1[max_idx] = list1[idx]
            #list1[idx] = list1[max_idx]
            list1[idx] = temp
        return list1


numbers = [12,-3,40,133,300,32,34,1,7]
print(numbers)
print("==================================")
sorted_numbers = selection_sort(numbers)
print(sorted_numbers)
sorted_numbers = selection_sort(numbers, reverse=False)
print(sorted_numbers)
sorted_numbers = selection_sort(numbers, reverse=True)
```

```
42 print(sorted_numbers)
```

```
[12, -3, 40, 133, 300, 32, 34, 1, 7]
==================================
[-3, 1, 7, 12, 32, 34, 40, 133, 300]
[-3, 1, 7, 12, 32, 34, 40, 133, 300]
[300, 133, 40, 34, 32, 12, 7, 1, -3]
```

Bubble sort

An example of a computer **algorithm** is **bubble sort**. This is a simple algorithm used for taking a list of jumbled up numbers and putting them into the correct order. The algorithm runs as follows:

1. Look at the first number in the list.
2. Compare the current number with the next number.
3. Is the next number smaller than the current number? If so, swap the two numbers around. If not, do not swap.
4. Move to the next number along in the list and make this the current number.
5. Repeat from step 2 until the last number in the list has been reached.
6. If any numbers were swapped, repeat again from step 1.
7. If the end of the list is reached without any swaps being made, then the list is ordered and the algorithm can stop.

Bubble sort example

This algorithm could be used to sort the following list:

3, 2, 4, 1, 5

The first loop of the algorithm would produce:

- **3**, **2**, 4, 1, 5 (2<3 so the two values are swapped)
- 2, **3**, **4**, 1, 5 (3<4 so the two values are **not** swapped)
- 2, 3, **4**, **1**, 5 (1<4 so the two values are swapped)
- 2, 3, 1, **4**, **5** (4<5 so the two values are **not** swapped)
- 2, 3, 1, 4, 5 (**First pass completed**)

Values were swapped so the algorithm needs to run again. The second loop of the algorithm would start with the final list and run again as follows:

- **2**, **3**, 1, 4, 5 (2<3 so the values are **not** swapped)
- 2, **3**, **1**, 4, 5 (1<3 so the values are swapped)
- 2, 1, **3**, **4**, 5 (3<4 so the values are **not** swapped)
- 2, 1, 3, **4**, **5** (4<5 so the values are **not** swapped)
- 2, 1, 3, 4, 5 (**Second pass completed**)

- **2**, **1**, 3, 4, 5 (1<2 so the values are swapped)
- 1, **2**, **3**, 4, 5 (2<3 so the values are **not** swapped)
- 1, 2, **3**, **4**, 5 (3<4 so the values are **not** swapped)
- 1, 2, 3, **4**, **5** (4<5 so the values are **not** swapped)
- 1, 2, 3, 4, 5 (**Third pass completed**)

Values were swapped so the algorithm needs to run again. This time there will be no swaps as the values are in order:

1, 2, 3, 4, 5

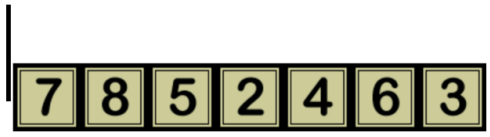
The algorithm has completed a loop without swapping anything and so it knows that the list is now ordered and can stop.

Selection Sort

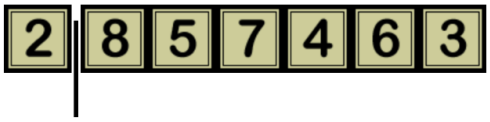
1. Get a list of unsorted numbers
2. Set a marker for the unsorted section at the front of the list
3. Repeat steps 4-6 until one number remains in the unsorted section
4. Compare all unsorted numbers in order to select the smallest one
5. Swap this number with the first number in the unsorted section
6. Advance the marker to the right by one position
7. Stop

1. First, we give the computer a list of unsorted numbers and store them in an array of memory cells.

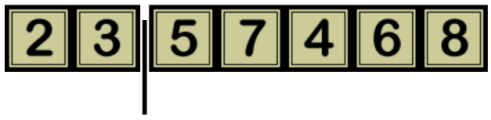
2. To begin the sort, the computer divides the sorted and unsorted sections of the list by placing a marker before the first number. To sort the numbers, the computer will repeatedly search the unsorted section for the smallest number, swap this number with the first number in the unsorted section, and update the sort marker.



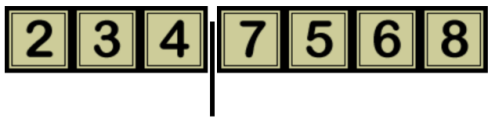
3. To find the smallest number in the unsorted section, the computer must make six comparisons: (7 < 8), (7 > 5), (5 > 2), (2 < 4), (2 < 6), and (2 > 3). After these comparisons, the computer knows that 2 is the smallest number, so it swaps this number with 7, the first number in the unsorted section, and advances the sort marker.



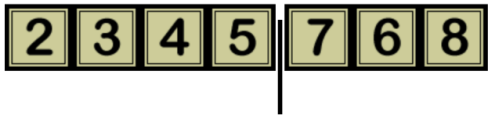
4. Now five more comparisons are needed to find the smallest number in the unsorted section: (8 > 5), (5 < 7), (5 > 4), (4 < 6), and (4 > 3). After these comparisons, the computer swaps 3, the smallest number in the unsorted section, with 8, the first number in the unsorted section, and advances the sort marker.



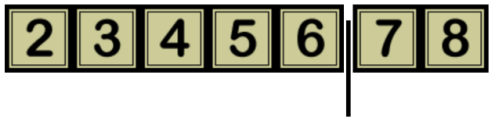
5. This time four comparisons are needed to determine that 4 is the smallest number in the unsorted section: (5 < 7), (5 > 4), (4 < 6), and (4 < 8). After these comparisons, the computer swaps 4 with 5 and then advances the sort marker.



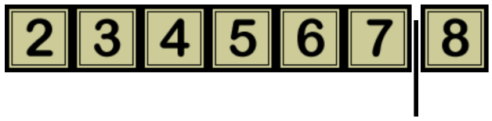
6. After three more comparisons, the computer identifies 5 as the smallest unsorted number: (7 > 5), (5 < 6), and (5 < 8). Then the computer swaps 5 with 7 and advances the sort marker.



7. This time only two comparisons are needed to determine that 6 is the smallest number: (7 > 6) and (6 < 8). After these two comparisons, the computer swaps 6 with 7 and then advances the sort marker.



8. Now we only need a single comparison to find the right position for 7: (7 < 8). Since 7 is the smallest number and it is also the first number in the unsorted section, the computer does not need to swap this number. It only needs to advance the sort marker. Now there is only one number in the unsorted section, so the list of numbers is sorted and the Selection Sort algorithm is complete.



In [1]:

```
#Linear Search

def func(list1, search):
    for i in list1:
        if i==search:
            return list1.index(i)
    return -1

list1 = [-13,23,14,1,4,5,-90]
search = 23
index = func(list1, search)
if index != -1:
    print("Element is in", index)
else:
    print("Not found")
```

```
Element is in 1
```

In [2]:

```
#Binary Search

def binarysearch(sortedSeq, x):
    l = 0
    r = len(sortedSeq)-1
    while l<=r:
        mid_index = (l+r)//2 #OR (l + (r-l)//2)

        if sortedSeq[mid_index]==x:
            return mid_index
        elif sortedSeq[mid_index]>x:
            r = mid_index - 1
        else:
            l = mid_index + 1

    return None
index = binarysearch([2,3,4,5,6,7,13,14,16], 4)
if index == None:
    print("Not Found")
else:
    print("Element at index", index)
```

```
Element at index 2
```

In [ ]:

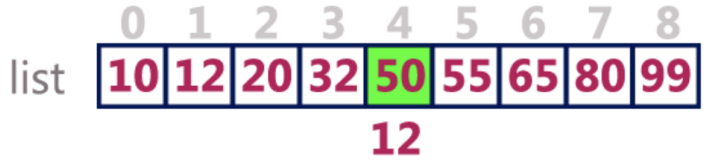
```

```

Binary Search

1. List must be in sorted order.
2. We need two pointers (low and high).
3. Search for a key element. In this case, 12.
4. Find mid.
5. If key element > mid value, search the right hand side.
   Change low pointer (mid+1).
6. If key element < mid value, search the left hand side.
   Change high pointer (mid-1).
7. If key element = mid value, stop.

Step 1:

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"