Using Google Colab

Read through the **Getting started** paragraph in the official documentation. You may also want to take a look into the **More resources** section.

To add a new code block or text block, use the buttons in the menus above or hover your mouse between two blocks, where more buttons should appear.

To execute all code blocks within this document, select `Runtime -> Run all`. When you submit your solution, this should run without throwing an error.

Sometimes, previously defined variables cause trouble in later executions. If this is the case, you can select `Runtime -> Restart runtime` to restart the runtime and erase all previous definitions.

Work Distribution

Initially, each member of the group worked on the assigment by themselves and then later we compared solutions in order to make a mutual decision on which version or combined versions to be presented in the final code. All members of the group have contributed in both code and discussion to the final submission equally.

Link to the assignment on colab: https://colab.research.google.com/drive/1-aUzeWcupfwc1G2EJhBoUKfpdmidlCpz?usp=sharing

Initialisation

This section contains code that is later used to test, measure and plot your implementations. You need to run it once to initialise it.

⇩ Click here to initialise the code.

↳ *1 cell har dolts*

Sorting Algorithms

In this exercise, you are going to implement the *Insertion Sort*, *Selection Sort* and *Sink Sort* algorithms.

- Each of your sorting functions should accept an unsorted list of numbers and sort it (in place). Of course, you are allowed to add as many helper functions as needed.
- Note that arrays (or lists) in the textbook are indexed from 1, whereas in Python (and most other programming languages) they are indexed from 0.
- Your implementation should have a good performance with respect to the asymptotic complexity found in the literature.

At the bottom of each code block, there is a function call to test your algorithm implementation. Run it to check if your implementation is correct.

Below the algorithms in this document, there is a code block that reports runtime statistics. Furthermore, at the end of the document, there are a couple of questions you should answer.

1. Insertion Sort (2 points)

Implement Insertion Sort as seen in the lecture.

```
def insertionsort(A):
    #loops through all the elements in the list, starting on second position
  for j in range(1,len(A)):
    #define the first element in the assumed unsorted list as 'key'
    key=A[j]
    #Represent the index of the last element in the sorted part of the list.
    i=j-1
    #compare the key element to each element in the sorted part
    while i >= 0 and A[i]> key:
      #Shifts larger elements to the right
      A[i+1]=A[i]
      #Move previous index to the sorted part
      i-=1
      #Insert the key to it's correct position
      A[i+1]= key
  return A


run_tests(insertionsort)
```

```
executing all tests on insertionsort...
Test 'empty list' passed!
Test 'one element' passed!
Test 'digits ascending' passed!
Test 'digits descending' passed!
Test 'digits shuffled' passed!
Test 'all entries the same' passed!
Test 'some duplicates' passed!
Test 'negative numbers' passed!
Test 'float numbers' passed!
Test 'mixed numbers' passed!
Test 'random numbers 0 (small)' passed!
Test 'random numbers 1 (small)' passed!
Test 'random numbers 2 (small)' passed!
Test 'random numbers 3 (small)' passed!
Test 'random numbers 4 (small)' passed!
Test 'random numbers 5 (medium)' passed!
Test 'random numbers 6 (medium)' passed!
Test 'random numbers 7 (medium)' passed!
Test 'random numbers 8 (medium)' passed!
Test 'random numbers 9 (medium)' passed!
Test 'random numbers 10 (large)' passed!
Test 'random numbers 11 (large)' passed!
Test 'random numbers 12 (large)' passed!
Test 'random numbers 13 (large)' passed!
Test 'random numbers 14 (large)' passed!
insertionsort (summary): 25/25 OK, 0/25 ERROR
```

2. Selection Sort (2 points)

Implement Selection Sort, which is a sorting algorithm that proceeds as follows:

- Throughout the execution, the array is divided into a sorted lower part and an unsorted upper part.
- Initially, the sorted lower part is empty. Upon termination, the sorted lower part spans the whole array.
- In each iteration of the algorithm:
    1. The algorithm searches for the minimal element of the unsorted upper part.
    2. This minimal element is swapped with the first element of the unsorted upper part of the array.
    3. The boundary between the sorted lower part and the unsorted upper part is raised by one to include the newly sorted element.

```
def selectionsort(A):
  #loops through all the elements in the list
  for i in range (len(A)):
    #Assume minimal value to be current element 'i'
    minimal_element_index=i
    # Find the minimum element in the remaining unsorted array
    for j in range (i+1, len(A)):
      # Compare the current element with the assumed minimum
      if A[j]< A[minimal_element_index]:
        # If a smaller element is found, update minimal_element_index
        minimal_element_index = j
    #Swap the found minimum element with the first element of the unsorted part
    A[i], A[minimal_element_index] = A[minimal_element_index], A[i]
  return A


run_tests(selectionsort)
```

```
executing all tests on selectionsort...
Test 'empty list' passed!
Test 'one element' passed!
Test 'digits ascending' passed!
Test 'digits descending' passed!
Test 'digits shuffled' passed!
Test 'all entries the same' passed!
Test 'some duplicates' passed!
Test 'negative numbers' passed!
Test 'float numbers' passed!
Test 'mixed numbers' passed!
Test 'random numbers 0 (small)' passed!
Test 'random numbers 1 (small)' passed!
Test 'random numbers 2 (small)' passed!
Test 'random numbers 3 (small)' passed!
Test 'random numbers 4 (small)' passed!
Test 'random numbers 5 (medium)' passed!
Test 'random numbers 6 (medium)' passed!
Test 'random numbers 7 (medium)' passed!
Test 'random numbers 8 (medium)' passed!
Test 'random numbers 9 (medium)' passed!
Test 'random numbers 10 (large)' passed!
Test 'random numbers 11 (large)' passed!
Test 'random numbers 12 (large)' passed!
Test 'random numbers 13 (large)' passed!
```

```
Test 'random numbers 14 (large)' passed!
selectionsort (summary): 25/25 OK, 0/25 ERROR
```

3. Sink Sort (2 points)

Implement Sink Sort, which is a sorting algorithm that proceeds as follows:

- Compare the first and second element of the array. If they are mismatched (the second element is smaller than the first one), swap them.
- Compare the second and third element of the array and swap them if they are mismatched.
- Continue until the end of the array, i.e. the second to last element and the last element.
- If you have swapped at least one pair of elements, repeat from the beginning.

```
def sinksort(lst):

    # Variables
    main_loop = True
    iteration = 1

    # Main loop
    while main_loop:

        # Variable for storing if list has been updated
        swapped = 0

        # Loop over list
        for i in range(len(lst)-1):

            # Compare neighbouring elements of the array and swap if smaller
            if lst[i+1] < lst[i]:
                temp = lst[i]
                lst[i] = lst[i+1]
                lst[i+1] = temp

                swapped = 1

        # Exit main loop if no changes during full iteration
        if swapped == 0:
            main_loop = False

        # Track iterations
        iteration +=1

    return lst



run_tests(sinksort)
```

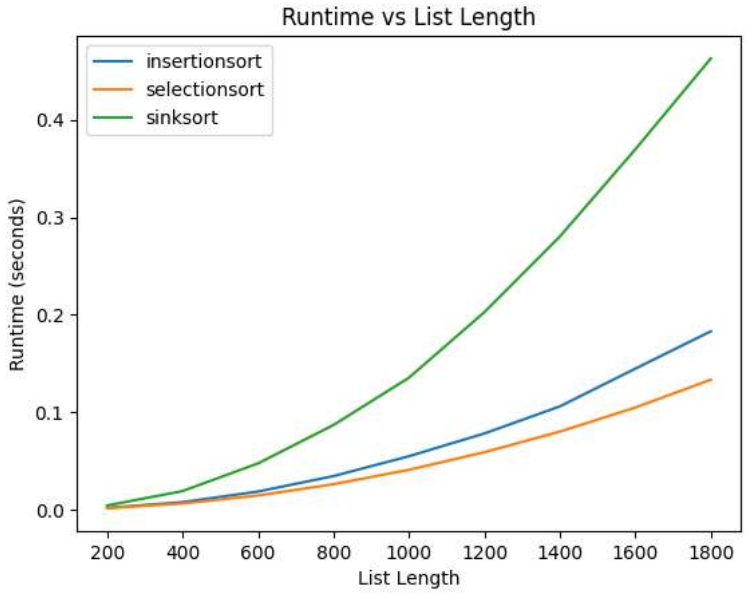
```
executing all tests on sinksort...
Test 'empty list' passed!
Test 'one element' passed!
Test 'digits ascending' passed!
Test 'digits descending' passed!
Test 'digits shuffled' passed!
Test 'all entries the same' passed!
Test 'some duplicates' passed!
Test 'negative numbers' passed!
Test 'float numbers' passed!
Test 'mixed numbers' passed!
Test 'random numbers 0 (small)' passed!
Test 'random numbers 1 (small)' passed!
Test 'random numbers 2 (small)' passed!
Test 'random numbers 3 (small)' passed!
Test 'random numbers 4 (small)' passed!
Test 'random numbers 5 (medium)' passed!
Test 'random numbers 6 (medium)' passed!
Test 'random numbers 7 (medium)' passed!
Test 'random numbers 8 (medium)' passed!
Test 'random numbers 9 (medium)' passed!
Test 'random numbers 10 (large)' passed!
Test 'random numbers 11 (large)' passed!
Test 'random numbers 12 (large)' passed!
Test 'random numbers 13 (large)' passed!
Test 'random numbers 14 (large)' passed!
sinksort (summary): 25/25 OK, 0/25 ERROR
```

4. Runtime Comparisons (2 points)

The following code will measure the runtime of your algorithm implementations.

Compare the reported runtimes. What do you observe? How do you explain your observations?

```
measure_runtime([insertionsort, selectionsort, sinksort])
plot_runtime([insertionsort, selectionsort, sinksort])
```



Answer:

The three sorting functions differ in terms of runtime. The gap in speed between the functions increase when the lists size increases. The reason for this has to do with the number of iterations, comparisions and swaps that each sorting functions has to do.

(Runtime durations for the algorithms can vary based on the machine they are run on. The numbers below are from a different run.)

Insertionsort: @800 = 0.03s, @1600 = 0.13s: 2x length -> 4.33x time

selectionsort: @800 = 0.0125s, @1600 = 0.055s: 2x length -> 4,4x time

sinksort: @800 = 0.06125s, @1600 = 0.26125s: 2x length -> 4.26x time

The runtimes seem to be quadratic, that is, O(n^2). However, the functions that describe the runtime for the different algorithms also include a constant to the quadratic term, which is why they have different slopes.

5. Modified Sink Sort (1 point)

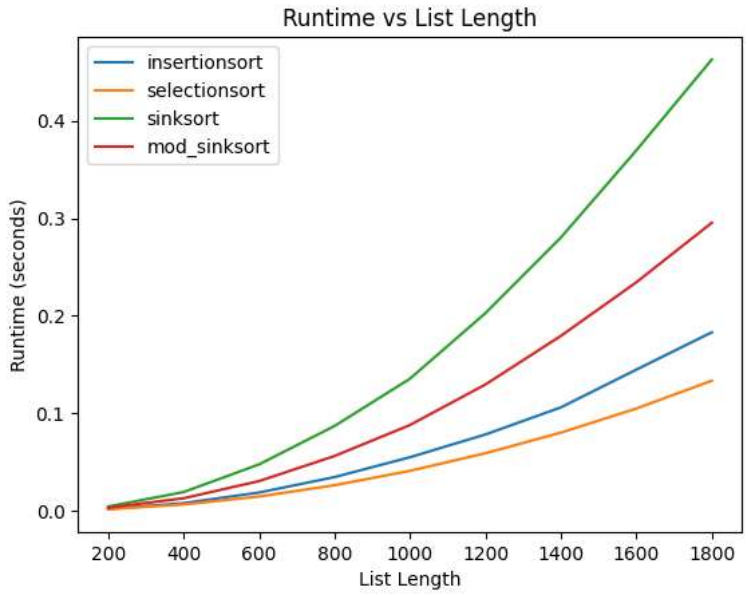
Have another look at the description of Sink Sort. Is it really necessary to go through the whole array in each iteration? Suggest an optimisation, implement it and compare the runtime to the unmodified version.

```
def mod_sinksort(lst):

    # Variables
    main_loop = True
    iteration = 1

    # Main loop
    while main_loop:

        # Variable for storing if list has been updated
        swapped = 0

        # Loop over list
        for i in range(len(lst)-iteration):                                    ###### MADE CHANGE HERE

            # Compare neighbouring elements of the array and swap if smaller
            if lst[i+1] < lst[i]:
                temp = lst[i]
                lst[i] = lst[i+1]
                lst[i+1] = temp

                swapped = 1
```

```
        # Exit main loop if no changes during full iteration
        if swapped == 0:
            main_loop = False

        # Track iterations
        iteration +=1

    return lst

run_tests(mod_sinksort)
```

```
executing all tests on mod_sinksort...
Test 'empty list' passed!
Test 'one element' passed!
Test 'digits ascending' passed!
Test 'digits descending' passed!
Test 'digits shuffled' passed!
Test 'all entries the same' passed!
Test 'some duplicates' passed!
Test 'negative numbers' passed!
Test 'float numbers' passed!
Test 'mixed numbers' passed!
Test 'random numbers 0 (small)' passed!
Test 'random numbers 1 (small)' passed!
Test 'random numbers 2 (small)' passed!
Test 'random numbers 3 (small)' passed!
Test 'random numbers 4 (small)' passed!
Test 'random numbers 5 (medium)' passed!
Test 'random numbers 6 (medium)' passed!
Test 'random numbers 7 (medium)' passed!
Test 'random numbers 8 (medium)' passed!
Test 'random numbers 9 (medium)' passed!
Test 'random numbers 10 (large)' passed!
Test 'random numbers 11 (large)' passed!
Test 'random numbers 12 (large)' passed!
Test 'random numbers 13 (large)' passed!
Test 'random numbers 14 (large)' passed!
mod_sinksort (summary): 25/25 OK, 0/25 ERROR
```

```
measure_runtime([mod_sinksort])
plot_runtime([insertionsort, selectionsort, sinksort, mod_sinksort])
```



Answer:

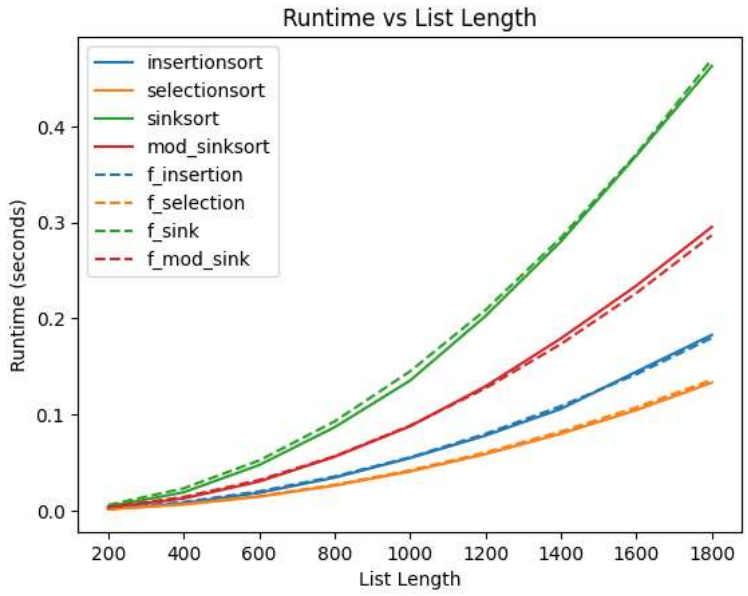
After each pass over the entire list, the largest element will be moved to the end of the list. This means that after each iteration we have one less element to check since we already know that the last element is the largest one.

6. Runtime approximations (1 point - hard question)

Using the statistics from the tests, for each algorithm, find a function `f(n)` which *approximately* describes the runtime of the algorithm in *seconds*, depending on the input size `n`. For example, if Insertion Sort takes 5 milliseconds to sort a list of 1000 numbers and 20 milliseconds to sort a list of 2000 numbers, then it should be that `f_insertion(1000)≈0.005` and `f_insertion(2000)≈0.020`.

It is possible to solve this by trial-and-error, but how would you get the answer by a simple calculation?

```
# All algorithms are on average O(n^2)


def f_insertion(size):
    c = 5.55e-8
    return c * size**2

def f_selection(size):
    c = 4.20e-8
    return c * size**2

def f_sink(size):
    c = 1.45e-7
    return c * size**2

def f_mod_sink(size):
    c = 8.85e-8
    return c * size**2


# plot all functions
for fun in [f_insertion, f_selection, f_sink, f_mod_sink]:
  runtimeDict[fun.__name__] = [fun(size) for size in sizes]
plot_runtime([insertionsort, selectionsort, sinksort, mod_sinksort], [f_insertion, f_selection, f_sink, f_mod_sink])
```



Answer:

(Runtime durations for the algorithms can vary based on the machine they are run on. The numbers below are from a different run.)

The mathematical way of getting there is by calculating the constant c, which determines the steepness of the quadratic function. For example, taking the sink sort algorithm, we can look at the runtime for a list length of 1400, which corresponds roughly to a runtime of 0.125 seconds. The formula will then be 0.125 = 1400^2 * c. The constant will roughly therefore be: c = 0.125/1400^2 = 6,38e-8. This can be done for all of the algorithms.