

Data Structures assignment

Andy Debrincat ICT1018

0135705L

This assignment was done using Python.

Question 1:

First the shell and quick sort algorithms were coded as follows:

```
def shell_sort(array):
    swaps = True
    length = len(array)
    jump = length // 2
    while jump > 0 and swaps:
        swaps = False
        for i in range(0, length - jump):
            if array[i] > array[i + jump]:
                swap(array, i, i + jump)
                swaps = True
        if jump != 1:
            jump //= 2

def quicksort(array, first, last):
    if first < last:
        left = first + 1
        right = last
        pivot_index = get_pivot_index(array, first, last)
        swap(array, pivot_index, first)
        while True:
            while array[left] < array[first] and left < last:
                left += 1
            while array[right] >= array[first] and right > first:
                right -= 1
            if left < right:
                swap(array, left, right)
            if left >= right:
                break

        swap(array, first, right)
        quicksort(array, first, right - 1)
        quicksort(array, right + 1, last)
```

These algorithms were inspired by pseudo-code given in the class PowerPoint, and work using the method explained in class.

This means that the shell sort works like the bubble sort but uses larger steps of $i + k$ always dividing k by 2 until it becomes a bubble sort when $k = 1$.

And the quick sort simply chooses a pivot and puts all the smaller elements to one side and the larger to the other. Then recursively calls itself on the two sub arrays separated by the pivot.

Note that the quicksort uses the `get_pivot_index()` function which works as follows:

```
def get_pivot_index(array, first, last):
    prime = 13

    if last - first > prime:
        temp = rand.sample(range(first, last), prime)
        counter = 1
        while counter < prime:
            item_to_compare = array[temp[counter]]
            item_index = temp[counter]
            sub_counter = counter-1
            while sub_counter > 0 and array[temp[sub_counter]] > item_to_compare:
                temp[sub_counter+1] = temp[sub_counter]
                sub_counter -= 1
            temp[sub_counter+1] = item_index
            counter += 1
        pivot = temp[prime // 2]
    else:
        pivot = (last + first) // 2
    return pivot
```

It selects a prime number of items from the given section of the array and sorts them using an insertion sort since it is a small array, and then picks the median as the pivot. This increases the likelihood of a good pivot with a very small overhead.

In addition to this a swap function was used which simply swaps two elements in an array given their indexes:

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

Then the requested arrays of random numbers were created:

```
sizeA = 256
sizeB = 200
A = np.empty(sizeA, dtype=int)
B = np.empty(sizeB, dtype=int)

for k in range(sizeA):
    A[k] = rand.randint(0, 1024)
    if k < sizeB:
        B[k] = rand.randint(0, 1024)

print("Array A :\n", A)
print("Sorted : ", is_sorted(A), "\n")
shell_sort(A)
print("Array A :\n", A)
print("Sorted : ", is_sorted(A), "\n")

print("Array B :\n", B)
print("Sorted : ", is_sorted(B), "\n")
quicksort(B, 0, len(B) - 1)
print("Array B :\n", B)
print("Sorted : ", is_sorted(B), "\n")
```

Note that **numpy** was used and imported as **np** and **random** was imported as **rand**.

In addition to this a function called **is_sorted** was used to check whether the arrays were sorted to check functionality.

```
def is_sorted(array):
    flag = True
    i = 1
    while i < len(array)-2 and flag:
        if (array[i] > array[i - 1] and array[i] > array[i + 1]) or (array[i] < array[i - 1] and array[i] < array[i + 1]):
            flag = False
        i += 1
    return flag
```

The way this function works uses the idea given in question 3 of extreme points.

When run the output was as follows:

```
Array A :
[ 275  87  592 152  717 149  242  664  659  740  134  547  266  912
 861 952 275 201  36 462 1019 690 262 757 161 270 611  63
1014 1011 687 845 111 106 468 196 307  96 682 290 514 856
117 661 309 940 27 151 963 478 631 964 48 503 542 245
154 1022 299 264 368 546 755 273 384 464 556 164 538 149
497  15 607 429  91 684 430 601 262 117 659 198 806 712
407 209 686  61 828 478 112 873 111 989 860 315 618 214
856 751 565 181 624 656 241 649 274 128 564 986 228 587
402 991 511 984 997 49 411 638 12 767 824 507 922 705
486 336 759 424 981 960 393 602 769 312 492 13 291 105
798 242 443 51 1013 442 913 381  3 441 559 517 367 41
 61 51 223 51 958 313 298 212 383 727 1015  9 31 657
871 539 167 895 239 759 441 665 110 416 344 566 110 309
303 669 67 836 577  6 978 203 961 207 927 63 506 1007
753 420 390 189 536 198 605 50 206 726 919 769 317 1003
 67 695 655 455 114 788 392 269 137 579 812 286 390 324
884 581 848 510 18 392 55 123 570 755 1011 157 544 571
990 108 179 952 712 502 745 804 83 196 407 392 285 312
576 499 678 683]
Sorted : False
```

```
Array A :
[  3  6  9 12 13 15 18 27 31 36 41 48 49 50
 51 51 51 55 61 61 63 63 67 67 83 87 91 96
105 106 108 110 110 111 111 112 114 117 117 123 128 134
137 149 149 151 152 154 157 161 164 167 179 181 189 196
196 198 198 201 203 206 207 209 212 214 223 228 239 241
242 242 245 262 262 264 266 269 270 273 274 275 275 285
286 290 291 298 299 303 307 309 309 312 312 313 315 317
324 336 344 367 368 381 383 384 390 390 392 392 392 393
402 407 407 411 416 420 424 429 430 441 441 442 443 455
462 464 468 478 478 486 492 497 499 502 503 506 507 510
511 514 517 536 538 539 542 544 546 547 556 559 564 565
566 570 571 576 577 579 581 587 592 601 602 605 607 611
618 624 631 638 649 655 656 657 659 659 661 664 665 669
678 682 683 684 686 687 690 695 705 712 712 717 726 727
740 745 751 753 755 755 757 759 759 767 769 769 788 798
804 806 812 824 828 836 845 848 856 856 860 861 871 873
884 895 912 913 919 922 927 940 952 952 958 960 961 963
964 978 981 984 986 989 990 991 997 1003 1007 1011 1011 1013
1014 1015 1019 1022]
Sorted : True
```

```
Array B :
[  24 905 421 827 910 124 997 277 694 880 981  83 690 519
994 242 421 783 468 1011 113 604 704 703 819 679 471 384
724 356 259 399 404 906 985 385 551 246 133 340 597 854
102 14 621 305 260 718 159 377 452 144 501 335 766 709
237 383 255 682 441 647 387 292 901 557 108 547 66 625
356 686 509 909 588 751 697 661 203 532 466 56 484 78
538 148 268 426 360 955 111 538 856 539 951 46 416 56
321 233 159 935 269 227 897 245 347 867 428 214 604 509
133 278 24 816 230 787 466 608 1014 57 847 526 765 452
100 276 722 635 949 554 923 918 518 613 265 467 812 805
878 114 80 533 42 568 410 498 518 1024 255 816 31 230
800 527 6 647 895 759 513 1010 622 382 617 0 1013 436
579 1 513 545 222 215 979 5 765 464 798 456 132 898
448 447 26 213 855 759 76 737 581 546 219 174 218 961
273 749 918 844]
Sorted : False
```

```
Array B :
[  0  1  5  6 14 24 24 26 31 42 46 56 56 57
 66 76 78 80 83 100 102 108 111 113 114 124 132 133
133 144 148 159 159 174 203 213 214 215 218 219 222 227
230 230 233 237 242 245 246 255 255 259 260 265 268 269
273 276 277 278 292 305 321 335 340 347 356 356 360 377
382 383 384 385 387 399 404 410 416 421 421 426 428 436
441 447 448 452 452 456 464 466 466 467 468 471 484 498
501 509 509 513 513 518 518 519 526 527 532 533 538 538
539 545 546 547 551 554 557 568 579 581 588 597 604 604
608 613 617 621 622 625 635 647 647 661 679 682 686 690
694 697 703 704 709 718 722 724 737 749 751 759 759 765
765 766 783 787 798 800 805 812 816 816 819 827 844 847
854 855 856 867 878 880 895 897 898 901 905 906 909 910
918 918 923 935 949 951 955 961 979 981 985 994 997 1010
1011 1013 1014 1024]
Sorted : True
```

The sorts work as expected.

Note that this code was run multiple times and the output was always satisfactory.

Question 2:

The requested merging algorithm was done exactly how the merge sort algorithm works. The pseudo code given in the power points was model for the algorithm. It works by iterating over both arrays simultaneously using two pointers and always picking the smallest. Finally, it adds the remaining elements of one of the arrays, depending on which one finished first. This is done in $O(n)$ steps.

```
def merge_arrays(arr1, arr2):  
  
    len_1 = len(arr1)  
    len_2 = len(arr2)  
  
    ret = np.empty((len_1 + len_2), dtype=int)  
    arr1_counter = 0  
    arr2_counter = 0  
    counter = 0  
  
    while arr1_counter < len_1 and arr2_counter < len_2:  
        if arr1[arr1_counter] < arr2[arr2_counter]:  
            ret[counter] = arr1[arr1_counter]  
            arr1_counter += 1  
        else:  
            ret[counter] = arr2[arr2_counter]  
            arr2_counter += 1  
        counter += 1  
  
    while arr1_counter < len_1:  
        ret[counter] = arr1[arr1_counter]  
        arr1_counter += 1  
        counter += 1  
  
    while arr2_counter < len_2:  
        ret[counter] = arr2[arr2_counter]  
        arr2_counter += 1  
        counter += 1  
  
    return ret
```

Then the two arrays of the previous question were merged into another array C.

```
C = merge_arrays(A, B)  
print("Array C :\n", C)  
print("Sorted : ", is_sorted(C), "\n")
```

The output was as follows:

```
Array C :  
[ 0 1 3 5 6 6 9 12 13 14 15 18 24 24  
 26 27 31 31 36 41 42 46 48 49 50 51 51 51  
 55 56 56 57 61 61 63 63 66 67 67 76 78 80  
 83 83 87 91 96 100 102 105 106 108 108 110 110 111  
 111 111 112 113 114 114 117 117 123 124 128 132 133 133  
 134 137 144 148 149 149 151 152 154 157 159 159 161 164  
 167 174 179 181 189 196 196 198 198 201 203 203 206 207  
 209 212 213 214 214 215 218 219 222 223 227 228 230 230  
 233 237 239 241 242 242 245 245 246 255 255 259 260  
 262 262 264 265 266 268 269 269 270 273 273 274 275 275  
 276 277 278 285 286 290 291 292 298 299 303 305 307 309  
 309 312 312 313 315 317 321 324 335 336 340 344 347 356  
 356 360 367 368 377 381 382 383 383 384 384 385 387 390  
 390 392 392 392 393 399 402 404 407 407 410 411 416 416  
 420 421 421 424 426 428 429 430 436 441 441 441 442 443  
 447 448 452 452 455 456 462 464 464 466 466 467 468 468  
 471 478 478 484 486 492 497 498 499 501 502 503 506 507  
 509 509 510 511 513 513 514 517 518 518 519 526 527 532  
 533 536 538 538 538 539 539 542 544 545 546 546 547 547  
 551 554 556 557 559 564 565 566 568 570 571 576 577 579  
 579 581 581 587 588 592 597 601 602 604 604 605 607 608  
 611 613 617 618 621 622 624 625 631 635 638 647 647 649  
 655 656 657 659 659 661 661 664 665 669 678 679 682 682  
 683 684 686 686 687 690 690 694 695 697 703 704 705 709  
 712 712 717 718 722 724 726 727 737 740 745 749 751 751  
 753 755 755 757 759 759 759 759 765 765 766 767 769 769  
 783 787 788 798 798 800 804 805 806 812 812 816 816 819  
 824 827 828 836 844 845 847 848 854 855 856 856 856 860  
 861 867 871 873 878 880 884 895 895 897 898 901 905 906  
 909 910 912 913 918 918 919 922 923 927 935 940 949 951  
 952 952 955 958 960 961 961 963 964 978 979 981 981 984  
 985 986 989 990 991 994 997 997 1003 1007 1010 1011 1011 1011  
 1013 1013 1014 1014 1015 1019 1022 1024]  
Sorted : True
```

A and B had been merged successfully.

Note that the code was run multiple times and always had a satisfactory output.

Question 3:

This algorithm was implemented by simply iterating through the given array and checking whether each element satisfies the conditions of an extreme point and printing it out. If it had no extreme points a flag is used to indicate that it is sorted.

The reason that having no extreme points implies the array is sorted is:

Suppose for all n in the range $(1, n-2)$ for an array of n elements, $A[n]$ is not an extreme point then for all n , $A[n-1] < A[n] < A[n+1]$ (ascending point) or $A[n-1] > A[n] > A[n+1]$ (descending point).

Then consider $A[1]$, by the hypothesis, $A[1]$ is either descending or ascending.

Case 1 : $A[1]$ is ascending means that $A[1] < A[2]$. But $A[2]$ is also not an extreme point which means that $A[2]$ is either descending or ascending. If $A[2]$ is descending then $A[1] > A[2] > A[3]$ which offers a contradiction since $A[1] > A[2]$ and $A[1] < A[2]$ is impossible.

Therefore $A[2]$ is also ascending.

Then if this is the base case, by induction if $A[n]$ is ascending then $A[n+1]$ is ascending.

This means that the whole array is in ascending order.

Case 2: Similarly if $A[1]$ is descending then the whole array is descending.

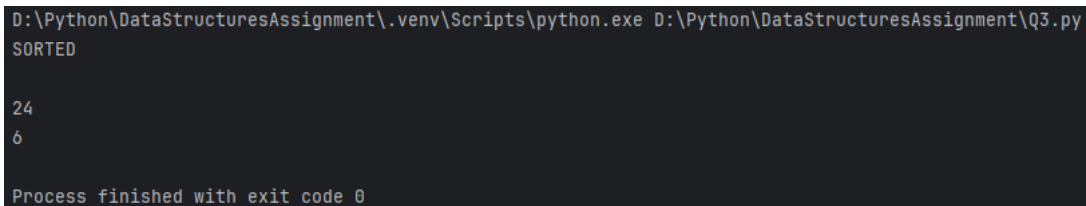
Therefore, in both cases the array is sorted.


```
def extreme_points(array):  
    is_sorted = True  
    for i in range(1, len(array) - 2):  
        if (array[i] > array[i - 1] and array[i] > array[i + 1]) or (array[i] <  
array[i - 1] and array[i] < array[i + 1]):  
            print(array[i])  
            is_sorted = False  
  
    if is_sorted:  
        print("SORTED\n")
```

The algorithm was tested using two arrays A and B, where A is sorted, and B is not. B's extreme points are 24 and 6.

```
A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
B = [1, 2, 3, 4, 24, 6, 7, 8, 9, 10]  
  
extreme_points(A)  
extreme_points(B)
```

The output was as follows :



```
D:\Python\DataStructuresAssignment\.venv\Scripts\python.exe D:\Python\DataStructuresAssignment\Q3.py  
SORTED  
  
24  
6  
  
Process finished with exit code 0
```

The output matches the expected outcome.

Question 4:

A function was made such that given an array it will return an array of pairs of pairs which satisfy the given conditions in the question.

```
def find_2_pairs(array):  
  
    array = list(set(array))  
    pairs = []  
    products = {}  
    n = len(array)  
  
    for i in range(n):  
        for j in range(i + 1, n):  
            product = array[i] * array[j]  
            if product not in products:  
                products[product] = []  
            if (array[i], array[j]) and (array[j], array[i]) not in  
products[product]:  
                products[product].append((array[i], array[j]))  
  
    for product in products:  
        length = len(products[product])  
        if length >= 2:  
            for i in range(length):  
                for j in range(i + 1, length):  
                    different = []  
                    if products[product][i][0] not in different :  
                        different.append(products[product][i][0])  
  
                    if products[product][i][1] not in different:  
                        different.append(products[product][i][1])  
  
                    if products[product][j][0] not in different:  
                        different.append(products[product][j][0])  
                    if products[product][j][1] not in different:  
                        different.append(products[product][j][1])  
  
                    if len(different) == 4:  
                        pairs.append((products[product][i], products[product][j]))  
  
    return pairs
```

First the function removes all duplicates from the given array by casting it into a set and back to a list.

Then it populates the dictionary called **products** with all the possible products in the array and the elements that yielded that product.

The dictionary acts as a one-to-many relation that maps the product to any two numbers that when multiplied yield that same product in the array.

Care was taken to make sure that there are no duplicate products or duplicate pairs that make a product or their symmetries to avoid repetition.

Finally, the **product** dictionary is traversed, and all those products having two or more pairs mapping onto it are taken in a binomial manner where if there are n pairs mapping onto a product, then there will be $n \text{ choose } 2$ pairs of pairs. These pairs of pairs will then be checked to obey the rule that no number must be equal in the pairs. This was done by checking all four of the elements of any one pair of pairs and if they are distinct, they are added to an array called **different**, and then it is checked to be sure that **different** has 4 elements, if not then the pair is not sufficient.

If a pair is sufficient it is added to the return list.

The function was tested using the following array:

```
A = [1, 1, 2, 4, 6, 24, 34, 36]
pairsA = find_2_pairs(A)

for pair in pairsA:
    print(pair)
```

The expected outcome would be the pairs of pairs **((1,24),(4,6))** and **((4,36),(6,24))** since no other pair of pairs matches the criteria.

The output was as follows:

```
D:\Python\DataStructuresAssignment\.venv\Scripts\python.exe D:\Python\DataStructuresAssignment\Q4.py
((1, 24), (4, 6))
((4, 36), (6, 24))

Process finished with exit code 0
```

This matches the expected output.

ChatGPT helped with the debugging process since the initial code kept giving lots of duplicate pairs. The idea of using a dictionary was taken from there.

Then a list with numbers from 1 till 1024 was given and in order to check validity two short functions were called that check whether a list of 2 pairs is valid by checking the conditions given and checking if the list has any duplicates through some permutation of a , b, c, d in ((a, b),(c, d)).

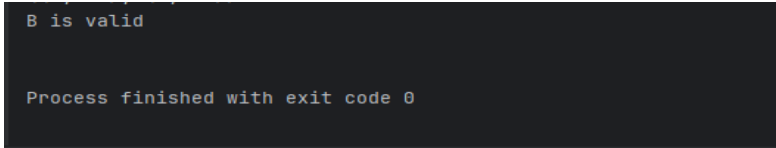
```
def is_valid(pairs):
    flag = True
    for pair in pairs:
        a, b = pair[0]
        c, d = pair[1]
        if a*b != c*d or (a == b) or (b == c) or (c == d) or (d == a):
            flag = False
    return flag

def has_duplicates(pairs):
    set_of_pairs = {frozenset(element for sub_tuple in pair for element in
sub_tuple) for pair in pairs}
    if len(set_of_pairs) == len(pairs):
        return False
    return True

B = list(range(1, 1025))
pairsB = find_2_pairs(B)

if is_valid(pairsB) and (not has_duplicates(pairsB)):
    print("B is valid\n")
else:
    print("B is invalid\n")
```

The output was printed.



```
B is valid

Process finished with exit code 0
```

Question 5:

The ADT stack was implemented by creating a function that takes in any string and attempts to evaluate it as if it is a valid RPN expression.

If it is then the correct answer will be output and if it is not, then an error message is displayed.

The expected RPN expressions are such that any two numbers must be delimited using a comma, and the only operators allowed are addition, multiplication, and their inverses.

Then it simply reads the string character by character and if there is a number with more than one digit it is read as a whole number using a buffer.

The rest simply acts as a normal stack by either pushing a number to stack or when an operation is encountered it pops the previous two items on stack and performs the operation and pushes the result on stack. Always adjusting the pointer to the next empty location.

```
def evaluate(expression):
    stack = []
    stack_pt = 0
    final_value = 0
    operators = ['+', '-', '*', '/']

    i = 0
    while i < len(expression):
        if expression[i].isdigit():
            digit_buffer = ''
            j = i
            while expression[j].isdigit() and j < len(expression):
                digit_buffer += expression[j]
                j += 1
            stack.append(int(digit_buffer))
            stack_pt += 1
            i = j - 1

        elif expression[i] in operators and len(stack) >= 2:
            if expression[i] == '+':
                final_value = stack[stack_pt - 2] + stack[stack_pt-1]
            elif expression[i] == '-':
                final_value = stack[stack_pt - 2] - stack[stack_pt-1]
            elif expression[i] == '*':
                final_value = stack[stack_pt - 2] * stack[stack_pt-1]
```

```

        elif expression[i] == '/':
            if stack[stack_pt - 1] == '0':
                print("Div 0")
            else:
                final_value = stack[stack_pt - 2] / stack[stack_pt-1]

                stack[stack_pt - 2] = final_value
                del stack[stack_pt-1]
                stack_pt -= 1

        elif expression[i] != ',':
            print("Invalid expression")
            return
        i += 1

    if len(stack) != 1:
        print("Invalid expression")
        return
    else:
        print(stack[0])

```

The stack was tested out using the following expressions:

```

# answer = 10
expres = "10,2,2+*4/"
evaluate(expres)
# answer = 3
expres = "1,2+3+2/"
evaluate(expres)
# answer = 90
expres = "100,50+30+2/"
evaluate(expres)
expres = "ashdashdlsad"
evaluate(expres)

```

The output was as follows:

```

D:\Python\DataStructuresAssignment\.venv\Scripts\python.exe D:\Python\DataStructuresAssignment\Q5.py
10.0
3.0
90.0
Invalid expression

Process finished with exit code 0

```

Question 6:

The prime-checking function works by trial division, meaning that given a number it checks whether it is divisible by all the numbers up to itself.

However a few optimisations were made :

1. If a number p is not prime it means that there exist two integers $m, n < p$ such that $mn = p$. Suppose for contradiction that both m and n are greater than square root p .
This means that $mn > p$.
This is a contradiction since $mn = p$.
This means that we need only check up to the square root of p .
2. All prime numbers are obviously odd, so we need only check odd numbers after 2.

```
def is_prime(n):  
    if n <= 1:  
        return False  
    prime = True  
    q = 2  
    while prime and q*q <= n:  
        if n % q == 0:  
            prime = False  
        else:  
            if q == 2:  
                q += 1  
            else:  
                q += 2  
    return prime
```

The two optimisations are shown above by the condition $q*q \leq n$ and by the increment of $q+=2$ instead of $q+=1$.

Then the sieve of Eratosthenes was implemented to find all the prime numbers less than a given number and returning them in a list.

This was done by creating a flag array from 0 till the requested number and True means they are prime while false means that they are not prime.

All the numbers are initially assumed to be prime and then the flag array is iterated over and each time it encounters a true value its sets all the multiples of the respective number to false.

By the end of the list, only the primes would be left.

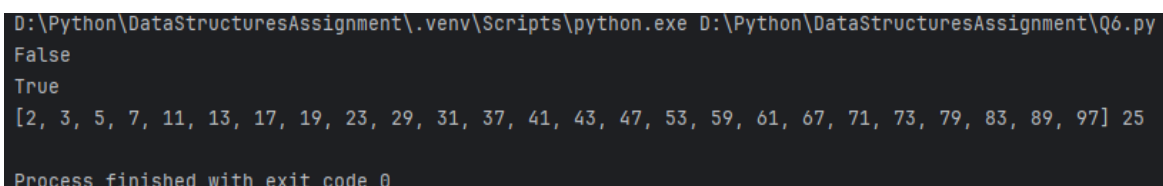
```
def sieve_of_eratosthenes(n):  
  
    primes = [True] * (n+1)  
    primes[0] = False  
    primes[1] = False  
    ret = []  
  
    for i in range(2, n):  
        if primes[i]:  
            ret.append(i)  
            j = 2*i  
            while j <= n:  
                primes[j] = False  
                j += i  
  
    return ret
```

The algorithms were tested with the following numbers:

```
print(is_prime(4))  
print(is_prime(1283))  
test = sieve_of_eratosthenes(100)  
print(test, len(test))
```

4 is not prime, 1283 is prime and there are 25 primes less than 100.

The output was as follows:



```
D:\Python\DataStructuresAssignment\.venv\Scripts\python.exe D:\Python\DataStructuresAssignment\Q6.py  
False  
True  
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97] 25  
  
Process finished with exit code 0
```

The algorithms work accordingly.

Question 7:

An algorithm to generate the Collatz sequence given a number was made and then a list was populated with the Collatz sequences from 2 to 512. These were then printed onto a CSV file in the same directory using the csv import in python.

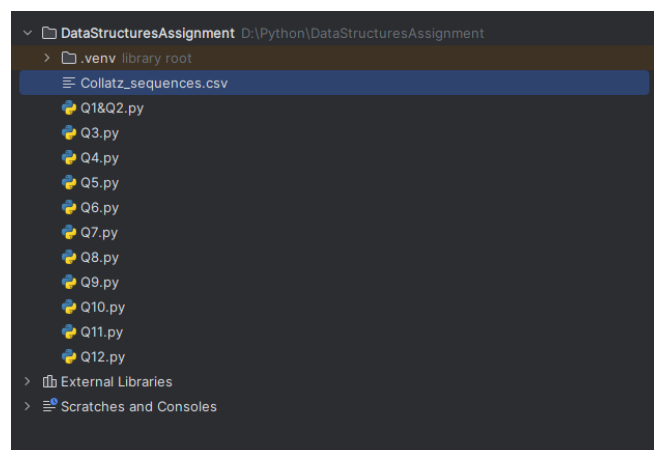
```
def collatz(n):
    ret = [n]
    temp = n
    while temp != 1:
        if temp % 2 == 0:
            temp //= 2
        else:
            temp = 3 * temp + 1
        ret.append(temp)
    return ret

# The collatz sequences of the numbers from 2-512 are then stored in a list which
# is then written to a csv file

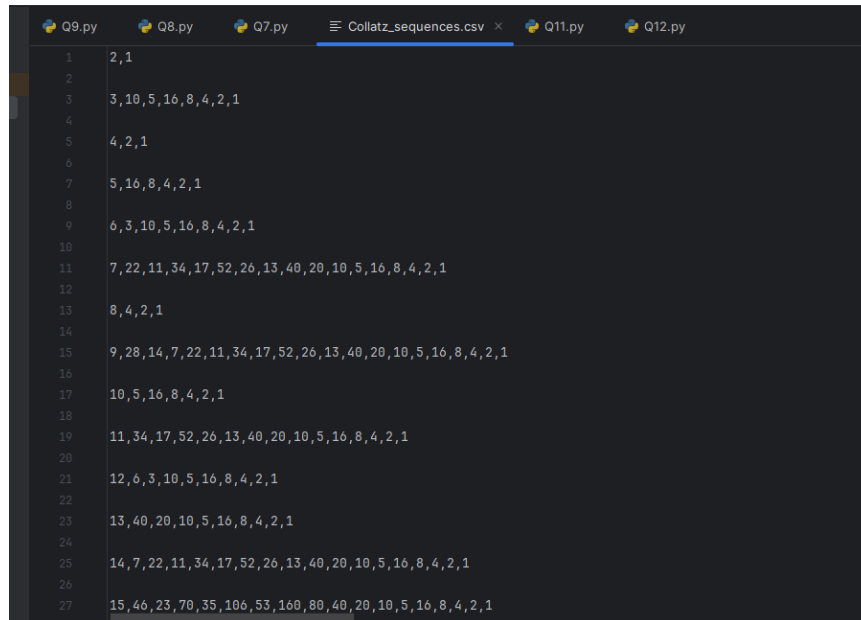
collatz_list = []
for i in range(2, 513):
    collatz_list.append(collatz(i))
filename = "Collatz_sequences.csv"

with open(filename, 'w') as csvfile:
    pointer = csv.writer(csvfile)
    pointer.writerows(collatz_list)
```

Every time the script is run a new file is created using write mode and all the sequences are written onto the file. The file is saved in the same directory where the executable is.



The file is too large to fit in one screenshot, but the beginning looks as follows:



```
1 2,1
2
3 3,10,5,16,8,4,2,1
4
5 4,2,1
6
7 5,16,8,4,2,1
8
9 6,3,10,5,16,8,4,2,1
10
11 7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1
12
13 8,4,2,1
14
15 9,28,14,7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1
16
17 10,5,16,8,4,2,1
18
19 11,34,17,52,26,13,40,20,10,5,16,8,4,2,1
20
21 12,6,3,10,5,16,8,4,2,1
22
23 13,40,20,10,5,16,8,4,2,1
24
25 14,7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1
26
27 15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1
```

Question 8:

To find an approximation of $\text{root}(n)$ the roots of $x^2 - n = 0$ are approximated using the newton Raphson method. Where $f(x) = x^2 - n$ and $f'(x) = 2x$. The roots of such an equation are either $-\text{root}(n)$ or $\text{root}(n)$.

The formula for the newton Raphson ($x_1 = x_0 - f(x_0)/f'(x_0)$) method becomes simple for this choice of $f(x)$, and it is $x_1 = x_0 - (x_0^2 - n)/2 x_0$. The function accepts the number of iterations and the initial guess x_0 .

```
def newton_raphson(n, iterations, x1):
    if iterations >= 300 or n<=0:
        print("Invalid input\n")
        return 0
    x2 = 0
    for i in range(iterations):
        # A simple check to avoid division by 0
        if x1 != 0:
            x2 = x1 - ((x1**2)-n)/(2*x1)
        else:
            print("error")
            return 0
        x1 = x2
    return abs(x2)
```

The function was tested to find $\text{square_root}(2)$ given two initial guesses.

```
# expected answer 1.414213562
print(newton_raphson(2, 20, -2))
print(newton_raphson(2, 20, 2))
```

The output was as follows as expected.

```
D:\Python\DataStructuresAssignment\.venv\Scripts\python.exe D:\Python\DataStructuresAssignment\Q8.py
1.414213562373095
1.414213562373095

Process finished with exit code 0
```

Question 9:

To find the duplicates in a given array, a flag array was used which keeps track of which numbers have already been mentioned in the array.

```
def find_duplicates(array):  
    flag = []  
    ret = []  
    for i in range(len(array)):  
        if array[i] in flag:  
            if array[i] not in ret:  
                ret.append(array[i])  
        else:  
            flag.append(array[i])  
    return ret
```

The function simply goes through the given array and for each element checks whether it has already been encountered through the flag array. It adds the duplicates only once to the return array.

This algorithm's memory efficiency is $O(n)$ and its time complexity depends on the search algorithm python uses when using the keyword in. In the worst case it is a linear search making it between $O(n^2)$ and $O(n)$ since at first the flag array is empty and would increase in size as the loop progresses to a maximum of n . The nested search only adds an extra n making it somewhere in the order of $O(n^{3/2})$ (not exact).

The array used to test it was as follows:

```
# expected answer : 1,2,8  
arr = [1, 1, 2, 2, 2, 3, 4, 5, 6, 7, 8, 8, 9, 10]  
print(find_duplicates(arr))
```

The output was as expected:

```
D:\Python\DataStructuresAssignment\.venv\Scripts\python.exe D:\Python\DataStructuresAssignment\Q9.py
[1, 2, 8]

Process finished with exit code 0
```

Question 10:

The recursive relation used for the algorithm `find_largest(array)` was as follows:

If $A_n = \{a_0, a_1, \dots, a_{n-1}\}$

then $\text{find_largest}(A_n) = \max(A[n-1], \text{find_largest}(A_{n-1}))$

and $\text{find_largest}(A_1) = a_0$

Once this relation had been found it was simple to implement as a function:

```
def find_largest(array):
    if len(array) == 1:
        return array[0]
    else:
        temp = array[len(array)-1]
        del (array[len(array) - 1])
        return max(temp, find_largest(array))
```

The function was tested on the following array:

```
# expected answer : 23
print(find_largest([13, 2, 4, 5, 9, 23]))
```

The output was as expected:

```
D:\Python\DataStructuresAssignment\.venv\Scripts\python.exe D:\Python\DataStructuresAssignment\Q10.py
23

Process finished with exit code 0
```

Question 11:

The first n terms of the Maclaurin expansion of sine and cosine were used to compute the value of $\cos(x)$ and $\sin(x)$.

However, since the terms contain large factorials, a special way of computing them was needed.

Since every term is in the form $(x^n)/n!$ the terms were calculated by multiplying the equivalent expression :

$(x/n) * (x/(n-1)) * \dots * (x/1)$.

These would all be terms much smaller than $n!$ making them easier to compute than a large factorial. This makes it possible to take in a large number of terms from the series expansion.

The function was made to take in the desired value of x , the number of desired terms ' n ' and the significant figures to round to ' s '.

```
def cos(x, n, s):
    ret = 0
    for i in range(n):
        temp = 1
        for j in range(1, 2*i+1):
            temp *= x/j

        ret += ((-1)**i)*temp
    return round(ret, s)

def sin(x, n, s):
    ret = 0
```

```

for i in range(n):
    temp = 1
    for j in range(1, 2 * i + 2):
        temp *= x / j

    ret += ((-1) ** i) * temp
return round(ret, s)

```

Then the functions were tested using `pi` as a parameter and a large number of terms that would not have been possible using normal factorial calculations.

```

number_of_terms = 1000
significant_figures = 7

# expected value : -1
print(cos(math.pi, number_of_terms, significant_figures))

# expected value : 0.707106
print(cos(math.pi/4, number_of_terms, significant_figures))

# expected value : 0
print(sin(math.pi, number_of_terms, significant_figures))

# expected value : 0.707106
print(sin(math.pi/4, number_of_terms, significant_figures))

```

The output was as expected, and only took less than 2 seconds.

```

D:\Python\DataStructuresAssignment\.venv\Scripts\python.exe D:\Python\DataStructuresAssignment\Q11.py
-1.0
0.7071068
0.0
0.7071068

Process finished with exit code 0

```

Note that the idea for computing large factorials was taken from the probability unit SOR1110.

Question 12:

In order to calculate the sum of the Fibonacci numbers up to n , first the Fibonacci number n was calculated using an iterative approach and in the same loop, the next number in the sequence was added to the sum, until n is reached.

```
def sum_fibonacci(n):
    if n == 1:
        return 1
    if n == 2:
        return 2
    if n < 1:
        return 0

    total = 2
    x_n_1 = 1
    x_n_2 = 1

    for i in range(3, n+1):
        temp = x_n_1 + x_n_2
        x_n_2 = x_n_1
        x_n_1 = temp
        total += temp
    return total
```

Then the function was tested:

```
# expected answer: 33
print(sum_fibonacci(7))
# expected answer: 0
print(sum_fibonacci(-2))
# expected answer: 4
print(sum_fibonacci(3))
```

The output was as expected:

```
D:\Python\DataStructuresAssignment\.venv\Scripts\python.exe D:\Python\DataStructuresAssignment\Q12.py
33
0
4

Process finished with exit code 0
```