MINISTERUL EDUCAȚIEI
INSPECTORATUL ȘCOLAR JUDEȚEAN BACĂU

COLEGIUL NAȚIONAL „GHEORGHE VRĂNCEANU”
STR. LUCREȚIU PĂTRĂȘCANU, NR. 30, BACĂU TEL. 0334/405941 FAX 0334/405942
email cnghvranceanu@yahoo.co.uk

Sorting Visualiser

LUCRARE PENTRU EXAMENUL DE ATESTARE A COMPETENȚELOR PROFESIONALE ÎN INFORMATICĂ

Candidat,
Bortoș Alexia-Ioana

Profesor îndrumător,
Avramescu Carmen

MAI 2023

Cuprins

Argument

"Computers are incredibly fast, accurate, and stupid. Human beings are incredibly slow, inaccurate, and brilliant. Together they are powerful beyond imagination."- Albert Einstein

Pasiunea pentru informatică a reprezentat mereu o constantă în viața mea. Deși, în aparență, informatica pare să se bazeze respectarea cu strictețe a unor tipare predefinite, în esență, aceasta oferă un cadru de gândire critică primordial în dezvoltarea abilității de a găsi soluții inovative la problemele întâmpinate în viața de zi cu zi.

Pentru a ajunge, însă, la performanța de a rezolva probleme ce presupun un grad ridicat de abstractizare, este necesară înțelegerea conceptelor ce stau la baza algoritmicii. Una dintre problemele clasice și cel mai frecvent întâlnite în informatică este sortarea unei liste de obiecte după anumite criterii.

Scopul acestui atestat este de a facilita înțelegerea unor noțiuni-cheie cu privire la algoritmii de sortare, printr-o aplicație interactivă ce ilustrează vizual modul în care se realizează comparațiile și interschimbările între elementele unui vector, în funcție de metoda aleasă. Cu ajutorul acestui *Sorting Visualiser*, utilizatorii pot analiza și realiza comparații între *Bubble Sort*, *Selection Sort* și *Insertion Sort* – trei algoritmi de sortare *„in place"* având complexitatea $O(n^2)$, care diferă, însă, în mod substanțial. Propunând o perspectivă aparte asupra unui concept de bază, aplicația le oferă utilizatorilor oportunitatea de a-și consolida cunoștințele sau de a-și corecta eventualele fisuri în raționament.

Despre Python

Python este un limbaj de programare dinamic, de nivel înalt, ce pune accent pe expresivitatea și înțelegerea ușoară a codului. Sintaxa sa permite implementări echivalente cu alte limbaje în mai puține linii de cod. Datorită acestui fapt, Python este foarte răspândit atât în programarea de aplicații, cât și în zona de scripting.

Limbajul facilitează mai multe paradigme de programare, în special paradigma imperativă (*C*) și pe cea orientată pe obiecte (*Java*). Spre deosebire de *C*, Python nu este un limbaj compilat, ci interpretat, ceea ce înseamnă că programele Python sunt transformate într-un limbaj intermediar. Acest lucru permite codului să fie ușor de portat pe diverse sisteme de operare și arhitecturi hardware.

Sintaxa este gândită în așa fel încât programele Python să fie ușor de citit. Acest lucru este obținut prin folosirea de cuvinte în locul semnelor (de exemplu, *and* în loc de *&&*) și prin includerea indentării în limbaj. Astfel, în Python nu se folosesc acolade (ca în *C/C++*, *Java*), ci blocurile de cod se delimitează prin *indentare*. Programele Python sunt, de multe ori, foarte aproape de o "implementare" echivalentă în pseudocod.

❖ Variabile

Python este un limbaj de programare în totalitate orientat pe obiecte, și nu "tastat static". Unul dintre avantajele utilizării limbajului Python este faptul că orice variabilă poate reține dinamic, pe parcursul unui program, valori de tipuri diferite, fără a impune tipul la declarare, precum inițializarea în *C++*. Fiecare variabilă în Python este considerată un obiect.

❖ Funcții

Funcțiile sunt o metodă convenabilă de partiționare a codului în blocuri organizate ce facilitează gestionarea acestuia. În Python, funcțiile sunt definite folosind cuvântul cheie **def**, urmat de numele funcției ca denumire pentru blocul de instrucțiuni care urmează. De exemplu:

```
def sumaNumerelor(a, b):
    return (a + b)
```

- **Funcții generatoare**

În Python, funcțiile generatoare sunt un tip special de funcții care crearea de *obiecte iterabile* personalizate într-un mod simplu și eficient. În loc să returneze o valoare și să își finalizeze execuția, o funcție generator poate să își suspende execuția după producerea unei valori, păstrându-și starea internă. La următorul apel, aceasta își reia execuția din punctul în care a rămas și produce următoarea valoare.

O funcție generator este definită utilizând cuvântul cheie ***yield*** în loc de ***return***. În timpul execuției, când se întâlnește o instrucțiune ***yield***, aceasta întrerupe temporar execuția funcției și returnează valoarea specificată ca argument. În următorul exemplu, funcția **numere_pare** este un generator care produce numere pare până la **n**.

```
def numere_pare(n):
    for i in range(n):
        if i % 2 == 0:
            yield i
# Apelarea funcției generator
generator = numere_pare(10)

# Iterarea prin valorile generate
for numar in generator:
    print(numar)
```

Utilizarea funcțiilor generatoare poate fi utilă atunci când se lucrează cu seturi mari de date sau când se dorește generarea de valori într-un mod eficient în ceea ce privește memoria.

❖ Obiecte și clase

Obiectele sunt o încapsulare de variabile și funcții într-o singură entitate. Obiectele își procură atributele și metodele din clase. **Clasele** sunt, în esență, niște șabloane pentru crearea obiectelor. O clasă este creată folosind cuvântul cheie ***class***, iar atributele şi metodele acesteia sunt listate într-un bloc indentat. Un exemplu clasic de clasă:

```
class ClasaMea:
    var = "hello"
    def functie(self):
        print ("Hello from the other side!")

obiectulMeu = ClasaMea()
print (obiectulMeu.var) # se va afișa conținutul variabilei "var"
```

```
obiectulMeu.functie() # acceseaza funcția din obiect, care va afișa mesajul
"Hello from the other side!"
```

Se pot crea oricâte obiecte diferite în cadrul aceleiași clase, acestea dobândind toate *trăsăturile* clasei respective (atribute și). Cu toate acestea, fiecare obiect în parte conține copii independente ale variabilelor definite în interiorul clasei. De pildă, dacă am defini un alt obiect al clasei “clasaMea”, acesta ar *moșteni* tot ceea ce era deja definit în interiorul clasei respective, însă ele pot fi modificate, fără a strica definiția inițială a clasei. Pentru clasa definită anterior, avem:

```
obiectulUnu = ClasaMea()
obiectulDoi = ClasaMea()

obiectulDoi.var = "bye"

print (obiectulUnu.var)
print (obiectulDoi.var)

# Outputul va fi:
# hello
# bye
```

Programarea orientată pe obiecte

Programarea orientată pe obiect (Programare Orientată Obiectual) este unul din cei mai importanți pași făcuți în evoluția limbajelor de programare spre o mai puternică abstractizare în implementarea programelor. Ea a apărut din necesitatea exprimării problemei într-un mod mai simplu, pentru a putea fi înțeleasă de cât mai mulți programatori. Astfel unitățile care alcătuiesc un program se apropie mai mult de modul nostru de a gândi decât modul de lucru al calculatorului. Ideea POO (Programare Orientată Obiectual) este de a crea programele ca o colecție de obiecte, unități individuale de cod care interacționează unele cu altele, în loc de simple liste de instrucțiuni sau de apeluri de proceduri

Deși tehnica se numește "Programare Orientată Obiectual", conceptul de bază al ei este Clasa. Clasa, pe lângă faptul că abstractizează foarte mult analiza/sinteza problemei, are proprietatea de generalitate, ea desemnând o mulțime de obiecte care împart o serie de proprietăți.

De exemplu: Clasa "floare" desemnează toate plantele care au flori, precum clasa "Fruct" desemnează toate obiectele pe care noi le identificam ca fiind fructe. Bineînțeles, în implementarea efectivă a programului nu se lucrează cu entități abstracte, precum clasele ci se lucrează cu obiecte, care sunt "instanțieri" ale claselor. Altfel spus, plecând de la exemplul de mai sus, dacă se construiește un program care să lucreze cu fructe, el nu va prelucra entitatea "fruct" ci va lucra cu entități concrete ale clasei "fruct", adică "măr", "pară", "portocală", etc. Instanțierea (trecerea de la clasă la obiect) presupune, deci, atribuirea unor proprietăți specifice clasei, astfel încât aceasta să indice un obiect anume, care se diferențiază de toate celelalte obiecte din clasă printr-o serie de atribute.

1. Principii de bază

Abstractizarea – Este posibilitatea ca un program să separe unele aspecte ale informației pe care o manipulează, adică posibilitatea de a se concentra asupra esențialului. Fiecare obiect în sistem are rolul unui “actor” abstract, care poate executa acțiuni, își poate modifica și comunica starea și poate comunica cu alte obiecte din sistem fără a dezvălui cum au fost implementate acele facilitați. Procesele, funcțiile sau metodele pot fi de asemenea abstracte, și în acest caz sunt necesare o varietate de tehnici pentru a extinde abstractizarea:

Încapsularea – numită și *ascunderea de informații*: Asigură faptul că obiectele nu pot schimba starea internă a altor obiecte în mod direct (ci doar prin metode puse la dispoziție de obiectul respectiv); doar metodele proprii ale obiectului pot accesa starea acestuia. Fiecare tip

de obiect expune o interfață pentru celelalte obiecte care specifică modul cum acele obiecte pot interacționa cu el.

Polimorfismul – Este abilitatea de a procesa obiectele în mod diferit, în funcție de tipul sau de clasa lor. Mai exact, este abilitatea de a redefini metode pentru clasele derivate. De exemplu pentru o clasă Figura putem defini o metodă arie. Dacă Cerc, Dreptunghi, etc. vor extinde clasa Figura, acestea pot redefini metoda arie.

Moștenirea – Organizează și facilitează polimorfismul și încapsularea, permițând definirea și crearea unor clase specializate plecând de la clase (generale) deja definite - acestea pot împărtăși (și extinde) comportamentul lor, fără a fi nevoie de a-l redefini. Aceasta se face de obicei prin gruparea obiectelor în clase și prin definirea de clase ca extinderi ale unor clase existente. Conceptul de moștenire permite construirea unor clase noi, care păstrează caracteristicile și comportarea, deci datele și funcțiile membru, de la una sau mai multe clase definite anterior, numite *clase de bază,* fiind posibilă redefinirea sau adăugarea unor date și funcții noi. Se utilizează ideea: ”Anumite obiecte sunt similare, dar în același timp diferite”. O clasă moștenitoare a uneia sau mai multor clase de bază se numește *clasă derivată*. Esența moștenirii constă în posibilitatea refolosirii lucrurilor care funcționează.

2. POO în Python

În Python, cuvântul „obiect” nu se referă neapărat la instanțierea unei clase. Clasele în sine sunt obiecte, iar, în sens mai larg, în Python toate tipurile de date sunt obiecte. Există tipuri de date care nu sunt clase: numerele întregi, listele, fișierele.

O metodă este o funcție definită în cadrul unei clase. Primul argument pe care îl primesc metodele este o instanță a clasei de care aparțin; în acest scop se folosește, la definirea metodei, cuvântul rezervat ***self***.

❖ Metode speciale

• Metoda specială ***__init__()*** este apelată la instanțierea clasei (crearea unui obiect de tipul clasei) și poate fi asemănată cu un constructor.

• Metoda specială ***__del__()*** este apelată când nu mai sunt referințe la acel obiect (mecanism de garbage collector) și poate fi asemănată cu un destructor.

• Instanțierea se face prin apelarea obiectului clasă, posibil cu argumente.

❖ Variabile de clasă, variabile de instanță

Partea de date, atributele clasei, nu sunt altceva decât variabile obişnuite care sunt legate de spațiile de nume ale claselor şi obiectelor. Asta înseamnă că aceste nume sunt valabile numai în contextul claselor şi obiectelor respective. Din acest motiv acestea sunt numite spaţii de nume. Există două feluri de câmpuri - variabile de clasă şi variabile de obiect/instanţă, care sunt clasificate în funcţie de proprietarul variabilei.

• Variabilele de clasă sunt partajate - ele pot fi accesate de toate instanţele acelei clase. Există doar un exemplar al variabilei de clasă şi când o instanţă îi modifică valoarea, această modificare este văzută imediat de celelalte instanţe.

• Variabilele de instanţă sunt proprietatea fiecărei instanţe a clasei. În acest caz, fiecare obiect are propriul exemplar al acelui câmp adică ele nu sunt relaţionate în niciun fel cu câmpurile având acelaşi nume în alte instanţe.

❖ **Moștenirea**

Programarea orientată pe obiecte le permite claselor sa moștenească stările și comportamentele comune din alte clase, ceea ce înseamnă crearea unei clase copil pe baza unei clase părinte. Clasa copil conține toate proprietățile și metodele clasei părinte, dar poate include și elemente noi. În cazul în care vreuna dintre noile proprietăți sau metode are aceeași denumire ca o proprietate sau metoda din clasa părinte, vor fi utilizate cele din clasa copil. Numele clasei părinte trebuie trecut între paranteze după numele clasei copil. Aceasta înseamnă că noua clasă va conține toate proprietățile și metodele clasei părinte. Mai trebuie însă redefinită funcția *__init__*.

❖ **Referințe și liste de obiecte**

Operatorul de atribuire funcționează diferit pentru obiecte. În cazul variabilelor scalare, dacă scriem variabila2 = variabila1 înseamnă că variabila 2 va prelua valoarea variabilei 1. În cazul obiectelor, dacă avem o atribuire instanta2 = instanta1, cele două variabile vor reprezenta referințe către același obiect. Cu alte cuvinte, dacă proprietățile obiectului instanta1 sunt modificate, aceasta modificare va fi vizibilă și în instanta2.

Despre Pygame

Pygame este un set de module Python conceput pentru dezvoltarea jocurilor video. Include grafică pe computer și biblioteci de sunet concepute pentru a fi utilizate cu limbajul de programare Python.

❖ Display-uri și Suprafețe

Pe lângă module, *Pygame* include și mai multe clase Python, care încapsulează concepte non-dependente de hardware. Una dintre acestea este *Surface* care, în forma sa cea mai rudimentară, definește o zonă dreptunghiulară pe care se poate desena. În *Pygame,* totul este privit pe un diplay creat de utilizator, care poate fi doar o fereastră sau întregul ecran. Display-ul este creat folosind ***.set_mode()***, care returnează o suprafață reprezentând partea vizibilă a ferestrei. Aceasta este suprafața pe care o primesc ca parametru funcții de desen precum ***pygame.draw.circle().***

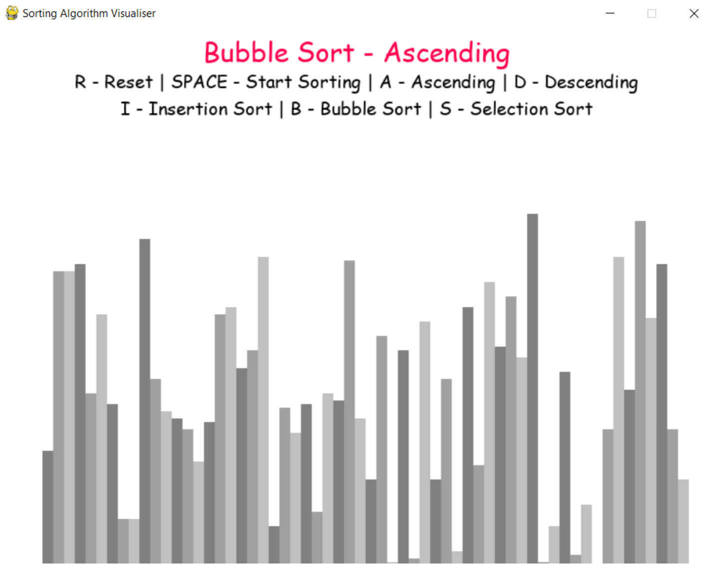
❖ Procesarea evenimentelor

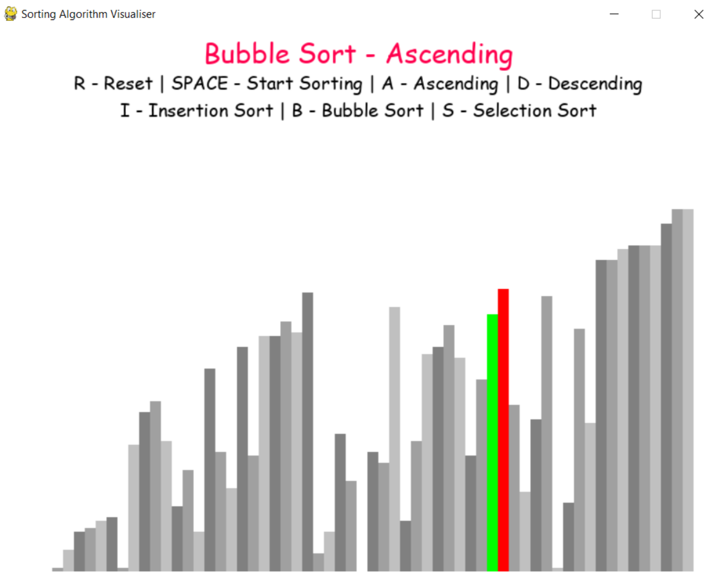
Apăsările de taste, mișcările mouse-ului și chiar mișcările joystick-ului sunt câteva dintre modalitățile prin care un utilizator poate furniza informații. Toate datele introduse de utilizator au ca rezultat generarea unui eveniment. Acestea se pot întâmpla în orice moment și adesea provin din afara programului. Toate evenimentele din *Pygame* sunt plasate în coada de evenimente, care poate fi apoi accesată și manipulată. Tratarea evenimentelor este denumită gestionarea lor, iar codul pentru a face acest lucru se numește *handler de evenimente*.

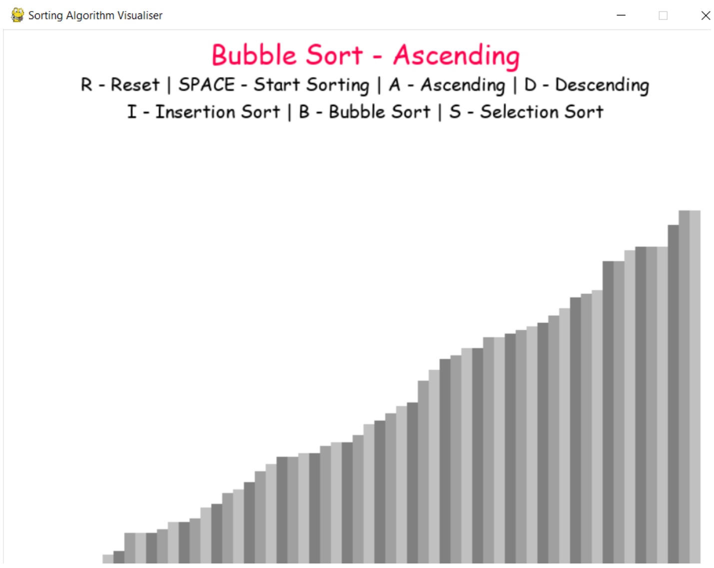
Evenimentele ce descriu apăsarea unor taste au tipul de eveniment KEYDOWN, iar evenimentul de închidere a ferestrei are tipul QUIT. Diferite tipuri de evenimente pot avea și alte date asociate. De exemplu, tipul de eveniment KEYDOWN are și o variabilă numită *tastă* pentru a indica ce tastă a fost apăsată. Accesarea listei tuturor evenimentelor active din coadă se realizează apelând ***pygame.event.get().*** Apoi se parcurge această listă, inspectându-se fiecare tip de eveniment și se răspunde în consecință:

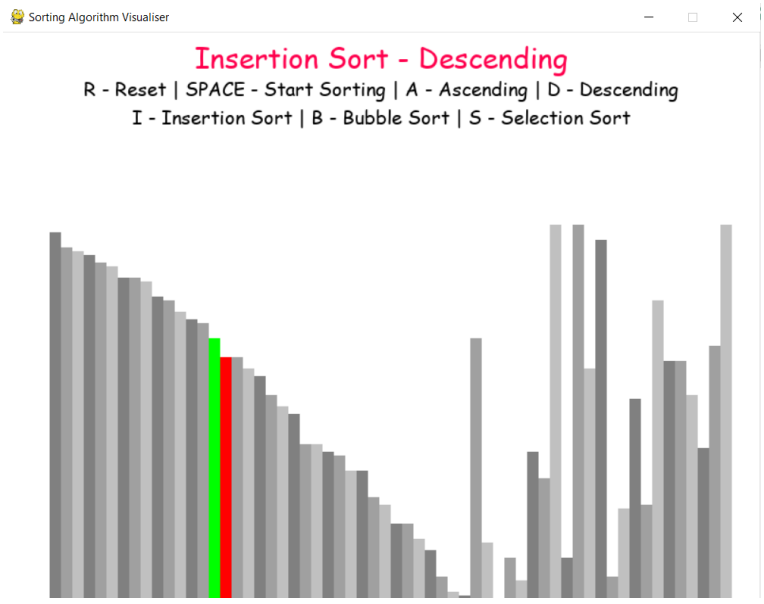
```
while running:
    for event in pygame.event.get():
        if event.type == KEYDOWN:
            if event.key == K_ESCAPE:
                running = False
        elif event.type == QUIT:
            running = False
```

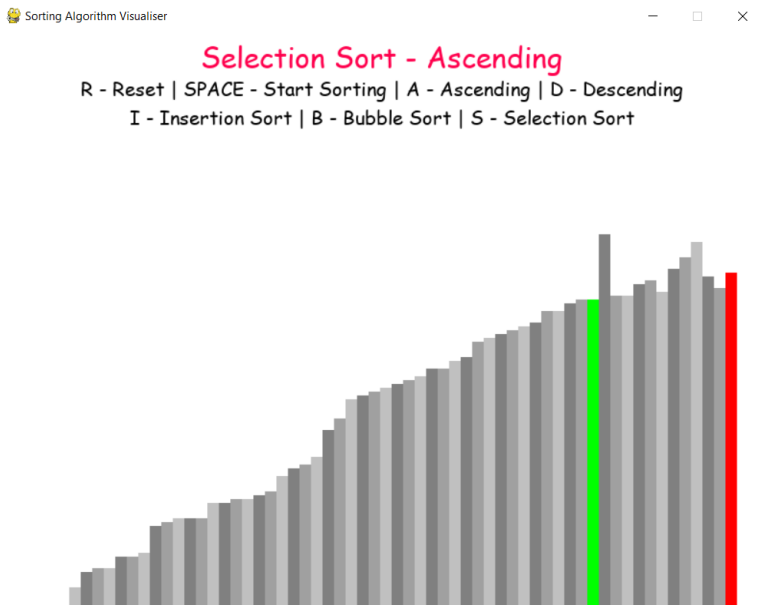
Interfața aplicației








Sorting Algorithm Visualiser
Insertion Sort - Descending
R - Reset | SPACE - Start Sorting | A - Ascending | D - Descending
I - Insertion Sort | B - Bubble Sort | S - Selection Sort


Sorting Algorithm Visualiser
Selection Sort - Ascending
R - Reset | SPACE - Start Sorting | A - Ascending | D - Descending
I - Insertion Sort | B - Bubble Sort | S - Selection Sort

Codul propriu-zis

*1. Definirea clasei **DrawInformation** – aceasta cuprinde toate informațiile necesare pentru desenarea pe ecran*

```
class DrawInformation:
    BLACK = 0, 0, 0
    WHITE = 255, 255, 255
    GREEN = 0, 255, 0
    RED = 255, 0, 0
    PINK = 255, 0, 77
    BACKGROUND_COLOR = WHITE

    GRADIENTS = [
        (128, 128, 128),
        (160, 160, 160),
        (192, 192, 192)
    ]

    FONT = pygame.font.SysFont('comicsans', 20)
    LARGE_FONT = pygame.font.SysFont('comicsans', 30)
    SIDE_PAD = 100
    TOP_PAD = 150

    def __init__(self, width, height, lst): # initializeaza fereastra aplicatiei cu dimensiunile specificate impreuna cu sirul de valori asociat
        self.width = width
        self.height = height

        self.window = pygame.display.set_mode((width, height)) # genereaza fereastra propriu-zisa
        pygame.display.set_caption("Sorting Algorithm Visualiser")
        self.set_list(lst)
```

```
    def set_list(self, lst): # calculeaza dimensiunile blocurilor in functie de dimensiunile ferestrei, lungimea vectorului si range-ul de valori
        self.lst = lst
        self.max_val = max(lst)
        self.min_val = min(lst)

        self.block_width = round((self.width - self.SIDE_PAD) / len(lst))
        self.block_height = math.floor((self.height - self.TOP_PAD) / (self.max_val - self.min_val))
        self.start_x = self.SIDE_PAD // 2
```

2. Implementarea funcțiilor care desenează pe ecran

```
def draw(draw_info, algo_name, ascending): # deseneaza antetul in functie de metoda de sortare aleasa de utilizator
    draw_info.window.fill(draw_info.BACKGROUND_COLOR)

    title = draw_info.LARGE_FONT.render(f"{algo_name} - {'Ascending' if ascending else 'Descending'}", 1, draw_info.PINK)
    draw_info.window.blit(title, (draw_info.width/2 - title.get_width()/2 , 5))

    controls = draw_info.FONT.render("R - Reset | SPACE - Start Sorting | A - Ascending | D - Descending", 1, draw_info.BLACK)
    draw_info.window.blit(controls, (draw_info.width/2 - controls.get_width()/2 , 45))

    sorting = draw_info.FONT.render("I - Insertion Sort | B - Bubble Sort | S - Selection Sort", 1, draw_info.BLACK)
    draw_info.window.blit(sorting, (draw_info.width/2 - sorting.get_width()/2 , 75))

    draw_list(draw_info)
    pygame.display.update()
```

```
def draw_list(draw_info, color_positions={}, clear_bg=False): # deseneaza pe ecran blocurile coresp valorilor numerice in trei nunate de gri
    lst = draw_info.lst

    if clear_bg: # reseteaza ecranul (doar partea cu blocurile)
        clear_rect = (draw_info.SIDE_PAD//2, draw_info.TOP_PAD,
                      draw_info.width - draw_info.SIDE_PAD, draw_info.height - draw_info.TOP_PAD)
        pygame.draw.rect(draw_info.window, draw_info.BACKGROUND_COLOR, clear_rect)

    for i, val in enumerate(lst):
        x = draw_info.start_x + i * draw_info.block_width
        y = draw_info.height - (val - draw_info.min_val) * draw_info.block_height

        color = draw_info.GRADIENTS[i % 3]

        if i in color_positions:
            color = color_positions[i]

        pygame.draw.rect(draw_info.window, color, (x, y, draw_info.block_width, draw_info.height))

    if clear_bg: # se redeseneaza ecranul pentru fiecare swap din timpul sortarii
        pygame.display.update()
```

3. Implementarea celor trei metode de sortare – acestea sortează vectorul și redesenează valorile pentru a ilustra interschimbările efectuate

```python
def generate_starting_list(n, min_val, max_val): # genereaza un sir de n valori aleatorii, cuprinse intre min_val si max_val
    lst = []

    for _ in range(n):
        val = random.randint(min_val, max_val)
        lst.append(val)

    return lst
```

```python
def bubble_sort(draw_info, ascending=True):
    lst = draw_info.lst

    for i in range(len(lst) - 1):
        for j in range(len(lst) - 1 - i):
            num1 = lst[j]
            num2 = lst[j + 1]

            if (num1 > num2 and ascending) or (num1 < num2 and not ascending):
                lst[j], lst[j + 1] = lst[j + 1], lst[j]
                draw_list(draw_info, {j: draw_info.GREEN, j + 1: draw_info.RED}, True)
                yield True

    return lst
```

```python
def insertion_sort(draw_info, ascending = True):
    lst = draw_info.lst

    for i in range(1, len(lst)):
        current = lst[i]

        j = i - 1
        while j >= 0 and ((current < lst[j] and ascending) or (current > lst[j] and not ascending)):
            lst[j + 1] = lst[j]
            j = j - 1
            draw_list(draw_info, {j: draw_info.GREEN, j + 1: draw_info.RED}, True)
            yield True
        lst[j + 1] = current

    return lst
```

```python
def selection_sort(draw_info, ascending = True):
    lst = draw_info.lst

    for i in range(len(lst)):
        k = i
        for j in range (i + 1, len(lst)):
            if(ascending and lst[j] < lst[k]):
                k = j
            elif(not ascending and lst[j] > lst[k]):
                k = j
        aux = lst[i]
        lst[i] = lst[k]
        lst[k] = aux
        draw_list(draw_info, {i: draw_info.GREEN, k: draw_info.RED}, True)
        yield True
    return lst
```

3. Implementarea funcției ***main***

```python
def main():
    run = True
    clock = pygame.time.Clock()

    n = 60
    min_val = 0
    max_val = 100

    lst = generate_starting_list(n, min_val, max_val) # sirul initial

    draw_info = DrawInformation(800, 600, lst)
    sorting = False
    ascending = True
    sorting_algorithm = bubble_sort
    sorting_algo_name = "Bubble Sort"
    sorting_algorithm_generator = None

    while run: # bucla infinita, pana cand este inchisa aplicatia
        clock.tick(60) #60 fps - viteza de executie

        if sorting:
            try:
                next(sorting_algorithm_generator)
            except StopIteration:
                sorting = False
        else:
            draw(draw_info, sorting_algo_name, ascending)
```

```python
        for event in pygame.event.get(): # se analizeaza fiecare actiune a utilizatorului
            if event.type == pygame.QUIT:
                run = False # inchide fereastra aplicatiei

            if event.type != pygame.KEYDOWN:
                continue

            if event.key == pygame.K_r: # tasta R (reset) genereaza un sir nou de valori
                lst = generate_starting_list(n, min_val, max_val)
                draw_info.set_list(lst)
                sorting = False
            elif event.key == pygame.K_SPACE and sorting == False: # tasta SPACE incepe sortarea
                sorting = True
                sorting_algorithm_generator = sorting_algorithm(draw_info, ascending)
            elif event.key == pygame.K_a and not sorting: # sortare crescatoare
                ascending = True
            elif event.key == pygame.K_d and not sorting: # sortare descrescatoare
                ascending = False
            elif event.key == pygame.K_i and not sorting:
                sorting_algorithm = insertion_sort
                sorting_algo_name = "Insertion Sort"
            elif event.key == pygame.K_b and not sorting:
                sorting_algorithm = bubble_sort
                sorting_algo_name = "Bubble Sort"
            elif event.key == pygame.K_s and not sorting:
                sorting_algorithm = selection_sort
                sorting_algo_name = "Selection Sort"


    pygame.quit()
```

Bibliografie

https://ro.wikipedia.org/wiki/Programare_orientat%C4%83_pe_obiecte

http://adrianabirlutiu.uab.ro/cursuri/BDOO/curs4_Python_Clase.pdf

https://ocw.cs.pub.ro/courses/ii/lab/laborator1

http://purepython.eaudeweb.ro/wiki/Cursuri/Programare-orientat%C4%83-pe-obiecte.html

https://realpython.com/pygame-a-primer/#background-and-setup