Візуалізація виконання алгоритму

Вступ

Візуалізація – це створення умов для візуального спостереження за програмною реалізацією алгоритму, який виконують на комп'ютері. Візуальне спостереження дає багато сигналів щодо поведінки алгоритму при виконанні кожного оператора програми. Таке спостереження потрібне завжди в процесі розробки і налагодження, доки не буде отримано остаточний варіант програми.

Спостереження можна реалізувати різними способами. Є дві категорії способів спостереження: *текстовий* в формі показу обчислених результатів на кожному кроці виконання алгоритму; *графічний* в формі малюнків, діаграм і анімацій процедури виконання кожного оператора програми. Текстовий спосіб спостереження можна реалізувати, по-перше, самому, шляхом додаткових операторів до основного алгоритму. По-друге, багато систем програмування мають вбудовані засоби за текстовою формою спостереження, які називають налагоджувачами (debugger).

Графічний спосіб спостереження вимагає спеціальної організації перетворення обчислених результатів до графічної форми, простіше кажучи, - до малюнка. Очевидно, що таке перетворення не може бути простим, а вимагає наявності окремих спеціалізованих графічних систем. Такі системи є, але вони працюють автономно і незалежно від нашої робочої системи програмування. Проте варто ознайомитись з графічними системами для демонстрації деяких типових алгоритмів задач.

Проста візуалізація в Python методом контрольних точок

Контрольна точка – це місце в програмі, яке потрібно перевірити щодо обчислених значень деяких величин. Це є оператор програми, важливий з точки зору власного впливу на подальше виконання програми і на остаточний результат, або важливий з точки зору контролю за ходом обчислень. Загальноприйнято вважати *контрольними точками* таке: початкове присвоєння значень величинам; читання з клавіатури або файлів значень величин; оператори галужень для з'ясування напряму наступних обчислень; оператори циклів; виклики процедур або функцій програміста; закінчення логічної частини обчислень.

Контрольні точки можна реалізувати вручну шляхом додавання до програми операторів `print()` тощо для запису поточних значень у контрольний файл або на екран.

Приклад 1. Алгоритм Евкліда. Програма без засобів контролю. Праворуч в рамці показані результати виконання.

```
# алгоритм Евкліда
a = int(input('перше число = '))
b = int(input('друге число = '))
# основний цикл
while a !=b :
    if a > b :  a=a-b
    else : b=b-a
# підсумкові операції
print('НСД = ', a)
pass
```

```
перше число = 48
друге число = 18
НСД =  6
```

Забігаючи вперед, зауважимо, що програма працює правильно, але для учнів важливо розуміти процес виконання програми. Для цього треба додати до програми контрольні точки наприклад, так, як показано далі.

Приклад 2. Алгоритм Евкліда. Програма з контрольними точками. Тут і далі оператори контрольних точок підкреслені.

```
# алгоритм Евкліда
a = int(input('перше число = '))
b = int(input('друге число = '))
# контрольна точка 1
print("Прочитані значення: ", a, b)
# основний цикл
while a !=b :
    if a > b :  a=a-b
    else : b=b-a
    # контрольна точка 2
    print("Ітерація цикла: ", a, b)
# підсумкові операції
print('НСД = ', a)
pass
```

```
перше число = 48
друге число = 18
Прочитані значення:  48 18
Ітерація цикла:  30 18
Ітерація цикла:  12 18
Ітерація цикла:  12 6
Ітерація цикла:  6 6
НСД =  6
```

Отже, тепер можна бачити динаміку виконання програми.

Візуалізація контрольними точками як функціями

Контрольні точки можна реалізувати як виклики функцій. В цьому разі отримуємо додаткові можливості спостереження. Проте треба скласти окремо функцію друкування контрольної точки.

Приклад 3. Речення складається з слів, записаних малими українськими буквами. Між словами один пропуск, в кінці речення – крапка. Надрукувати: 1) список букв, які зустрічаються в реченні не менше двох разів; 2) список букв, які зустрічаються в реченні рівно по одному разу; 3) список букв, які відсутні в реченні.

Задачу розв'язуємо за допомогою множин set, які дозволяють ефективно організувати операції додавання чи викреслення елементів і перевірки входжень елементів в множину.

```
# функція друкування контрольної точки
def content(controlnum,*obj,inline=True) :
    # controlnum - номер контрольної точки
    # obj - об'єкт друкування (кортеж)
    # inline=True - в рядок, False - в стовпчик
    print(controlnum, end=":  ")
    if inline :  print(*obj, sep="  ")
    else :  print(*obj, sep="\n")
    pass
# ---------------------------------------------
# Задача. Переліки українських букв в реченні.
sentence = "абв ба д." # задане речення
A = set()    # множина А
B = set()    # множина Б
# обидві множини на початку порожні
content(1,sentence,A,B)  # КТ1
for c in sentence :   # перегляд речення
    if c not in A : A.add(c)
    else : B.add(c)
    content(2,sentence,c,A,B)  # КТ2
# викреслити пропуск і крапку
A.discard(' ');  A.discard('.')
B.discard(' ');  B.discard('.')
```

```
content(3,sentence,A,B)  # КТ3
print(sorted(list(B)))
print(sorted(list(A-B)))
# загальний список українських літер
ukrlow = 'абвгґдеєжзиіїйклмнопрстуфхцчшщьюя'      # ґ - з Word
print(sorted(list(set(ukrlow)-A)))
pass
```

Протокол отриманих результатів:

```
1:  абв ба д.  set()  set()
2:  абв ба д.  а  {'а'}  set()
2:  абв ба д.  б  {'б', 'а'}  set()
2:  абв ба д.  в  {'в', 'б', 'а'}  set()
2:  абв ба д.     {'в', 'б', 'а', ' '}  set()
2:  абв ба д.  б  {'в', 'б', 'а', ' '}  {'б'}
2:  абв ба д.  а  {'в', 'б', 'а', ' '}  {'б', 'а'}
2:  абв ба д.     {'в', 'б', 'а', ' '}  {'б', 'а', ' '}
2:  абв ба д.  д  {'в', 'б', 'д', ' ', 'а'}  {'б', 'а', ' '}
2:  абв ба д.  .  {'в', '.', 'б', 'д', ' ', 'а'}  {'б', 'а', ' '}
3:  абв ба д.  {'в', 'б', 'д', 'а'}  {'б', 'а'}
['а', 'б']
['в', 'д']
['г', 'е', 'ж', 'з', 'и', 'й', 'к', 'л', 'м', 'н', 'о', 'п', 'р', 'с',
'т', 'у', 'ф', 'х', 'ц', 'ч', 'ш', 'щ', 'ь', 'ю', 'я', 'ґ', 'є', 'і',
'ї']
```

Контрольна точка з паузою

В попередніх прикладах ціла програма виконувалась без зупинок від початку до кінця. Протокол результатів виконання разом з даними контрольних точок можна вивчати після закінчення виконання програми. Проте може бути потреба в дотриманні паузи між тактами виконання програми. Тобто, ми б хотіли призупинити виконання програми після друкування контрольної точки, переглянути проміжні результати, оцінити ситуацію, після цього продовжити виконання далі.

Для цього в нашому випадку достатньо додати до функції друкування контрольної точки оператори, які б виконали таку затримку. Можна просто вставити оператор фіктивного читання даних з клавіатури до натиснення на клавішу Enter. Наприклад:

```
# функція друкування контрольної точки з паузою
def content(controlnum,*obj,inline=True,pause=False) :
    # controlnum - номер контрольної точки
    # obj - об'єкт друкування (кортеж)
    # inline=True – друкувати в рядок, False - в стовпчик
    # pause=False - без зупинки, True - чекати натиснення Enter
    print(controlnum, end=":  ")
    if inline :  print(*obj, sep="  ")
    else :  print(*obj, sep="\n")
    if pause: input()  # чекати натиснення Enter
    pass
```

Тепер виклик функції за потреби зробити паузу може виглядати так:

```
content(2,sentence,c,A,B,pause=True)  # КТ2
```

Метод стеження

Стеження - це метод візуалізації, при якому обираємо деяку змінну величину і стежимо за її значеннями впродовж всього часу виконання програми. Стеження можна виконати за декількома змінними величинами зразу, проте обирати надто багато величин не рекомендують, бо потрібно аналізувати дуже велику кількість інформації. Стеження за значеннями полягає в отриманні інформації про місце зміни значення і про конкретне нове отримане значення величини. Реалізувати метод стеження можна, використовуючи контрольні точки програми, - прості або функційні, - при кожному присвоєнні значення величині. Присвоєння відбувається як власне за оператором присвоєння, так і за іншими операторами, наприклад, при читанні з файла чи клавіатури, при поверненні з функції, тощо. Фактично цей метод є різновидом застосування контрольних точок.

Приклад 4. Чи є задане ціле число паліндромом? Число називають паліндромом, якщо перестановка цифр в оберненому порядку дає таке саме число, наприклад: 282, 3003, 242434242, 9.

Ідея розв'язку: відсікати по черзі від числа молодшу цифру і використовувати її для побудови нового числа за формулою "(попереднє відсічене * 10) + остання відсічена цифра". В кінці порівняти побудоване число з початковим.

```
# чи задане число є паліндромом
number = abs(int(input('Ціле число = ')))   # задано
N = number    # копія числа для відсікання цифр
newnumber = 0    # будемо будувати нове обернене число
content(1,newnumber,N)  # КТ1
# цикл відсікання цифр і будови нового числа
while N>0 :
    newnumber = newnumber*10 + N % 10
    N //= 10      # зменшити в 10 разів (націло)
    content(2,newnumber,N)  # КТ2
# друкуємо відповідь
print('є паліндромом') if number == newnumber else print('не паліндром')
content(3,newnumber,number)  # КТ3
pass
```

Протокол результатів:

```
Ціле число = 24542
1:  0  24542
2:  2  2454
2:  24  245
2:  245  24
2:  2454  2
2:  24542  0
є паліндромом
3:  24542  24542
```

Приклад 5. Задано додатнє ціле число p. Визначити, чи воно є простим.

Ідея розв'язку: ділити p по черзі на числа більші 1 і не більші p//2; число просте, якщо не ділиться на жоден вказаний дільник, тобто всі остачі не повинні дорівнювати нулю.

```
# перевірка числа - чи є простим
p = int(input('Ціле число більше 1 : '))
x = p // 2              # дільники до половини значення числа
content(1,x, p%x)  # КТ1
while x > 1 :
    if p % x == 0 :     # залишок ділення націло
        print(p, 'має дільник', x, '- не є простим')
        break           # перейти через блок else
    x -= 1              # зменшити на одиницю
    content(2,x, p%x)  # КТ2
else:                   # нормальне закінчення циклу
    print(p, 'число просте')
pass
```

Протокол результатів:

```
Ціле число більше 1 : 27
1:  13  1
2:  12  3
2:  11  5
2:  10  7
2:  9  0
27 має дільник 9 - не є простим
```

Метод логічної прокрутки - трасування

Логічна прокрутка – це метод візуалізації програми, при якому визначаємо шлях виконання операторів програми для кожного конкретного тесту. Шлях виконання залежить від операторів, які мають керований вихід (продовження різними шляхами): if-else, if-elif, while, while-else, for, for-else, break, continue, def-функції, try-except-finally, raise, assert, і т.п. Залежно від конкретних поточних значень величин може бути виконаний той чи інший оператор за керуючим виходом і відповідно отриманий інший результат. Логічна прокрутка дозволяє перевірити, чи правильно побудовані умови керуючих операторів, чи враховані всі комбінації вхідних даних, чи правильно заплановано оператори опрацювання даних для кожної умови керуючого оператора, чи перевірені всі лінійні ділянки програми і всі виходи керуючих операторів на сукупності тестів. Реалізувати логічну прокрутку можна шляхом комбінування контрольних точок, покрокового виконання програми і належної будови перегляду поточних значень величин програми.

Приклад 6. Задано текст алгебраїчної формули в лінійній формі запису, який включає круглі дужки. Вважати, що текст складений коректно, тобто, до кожної відкриваючої дужки є закриваюча і навпаки.

Надрукувати попарно позиції відповідних відкриваючих і закриваючих дужок

Алгоритм. Переглядаємо текст формули зліва направо. Номери позицій відкриваючих дужок записуємо в стек. Якщо знаходимо закриваючу дужку, тоді вершина стеку повинна мати позицію відповідної відкриваючої, яку читаємо і друкуємо пару номерів позицій.

Нижче подано текст частини класу стеку, яка приймає участь в обчисленнях, і текст розв'язку задачі з додатковими операторами трасування (контрольними точками). Функція `content()` така сама, як в попередньому прикладі – тут не друкуємо.

```
class Stack :
    def __init__ (self) :   # спочатку стек завжди пустий
        self.stack = []

    def __str__ (self) :  # список елементів стеку - для контролю
        return  "<bottom> [" + ",".join(map(str,self.stack)) + "] <top>"

    def push(self,obj) :  # записати в стек
        self.stack.append(obj)
        return self  # для кратних записів

    def pop(self) :  # читати з вершини
        if self.stack : return self.stack.pop()
        else : return None  # якщо стек пустий
    # інші методи стека . . .

# -------------- задача -----------------------
formula = "***(**(***)****(***)*(****(*))**)**"
s = Stack()
content(1, "Початок - пустий стек")
for k in range(len(formula)) :  # перегляд формули за літерами
    if formula[k] == "(" : s.push(k); content(2,"push (", k)
    elif formula[k] == ")" : content(3,"pop )",k); print(s.pop(),'-',k);
    # інші літери до уваги не приймаємо
pass
```

Протокол результатів:

```
1:  Початок - пустий стек
2:  push (  3
2:  push (  6
3:  pop )  10
6 - 10
2:  push (  15
3:  pop )  19
15 - 19
2:  push (  21
2:  push (  26
3:  pop )  28
26 - 28
3:  pop )  29
21 - 29
3:  pop )  32
3 - 32
```

Зауважимо, що метод трасування є подібний до методу стеження.

Візуалізація алгоритму в середовищі C++ додатковими операторами

Загальні принципи візуалізації, викладені вище для Python, можна використати для програм мовою C++. Але є деякі відмінності в реалізації методів візуалізації.

Мова C++ є мовою компільованого типу і має свої особливості будови. Наприклад, кожна величина програми C++ має наперед визначений тип. Так само кожний параметр функції належить фіксованому типу. В мові Python самі величини типу не мають, тому є поліморфними. Зокрема, фактичним параметром функції можна задати будь-яку величину і навіть цілий список величин, що ми й робили для функції `content()`:

```
  def content(controlnum,*obj,inline=True,pause=False)
```

Тут параметр `*obj` позначає будь-яку групу величин, тобто групу фактичних параметрів чи об'єктів.

Для мови C++ пряма реалізація такого підходу неможлива. Тому функції контролю, подібні до `content()`, можна реалізувати для одного чи декількох параметрів, але наперед визначеного типу. Отже, в разі застосування методів контрольних точок треба обмежити візуалізацію наперед заданим типом величини, або скласти декілька перевантажених варіантів функції `content()`.

Покажемо реалізацію методу простого стеження без використання функцій контролю. Якщо функція контролю потрібна – її можна реалізувати за правилами мови C++.

Приклад 7. В заданому масиві чисел знайти найбільше значення серед від'ємних. Зважити, що від'ємних чисел може не бути.

```
#include  <iostream>
using  namespace  std;
int main()
{
  int num[] = { 15,-9,0,32,-7,22,11,-4,-10,20 };
  int N = sizeof num / sizeof(int);  // кількість чисел в масиві
  // спочатку знайти найперше за порядком найменше, якщо таке є
  bool find = false;  int negative = 0;
  /*watch-1*/  cout << "watch-1: " << negative << endl;
  for (int i = 0; i < N && !find; i++)
  {
    if (num[i] < 0) { negative = num[i]; find = true; }
    /*watch-2*/  cout << "watch-2: " << negative << endl;
  }
  // перегляд масиву і пошук від'ємних, якщо такі є
  if(find)
  {
    for (int k = 0; k < N; k++)
    {
      if (num[k]<0 && num[k]>negative) negative = num[k];
      /*watch-3*/  cout << "watch-3: " << negative << endl;
    }
    cout << "max among negative is " << negative << endl;
  }
  else { cout << "numbers negative not find.\n"; }
  system("pause"); return 0;
}
```

Контрольні точки стеження позначені в програмі як /*watch-1*/ за різними номерами і зберігаються як коментарі. Додаткові оператори для контрольних точок тут підкреслені.

Протокол результатів:

```
watch-1: 0
watch-2: 0
watch-2: -9
watch-3: -9
watch-3: -9
watch-3: -9
watch-3: -9
watch-3: -7
watch-3: -7
watch-3: -7
watch-3: -4
watch-3: -4
watch-3: -4
max among negative is -4
```

Візуалізація в середовищі С++ системними засобами

Розглядаємо ту саму попередню задачу. Покажемо, як виконати візуалізацію, використовуючи вбудовані засоби середовища програмування С++. В цьому разі непотрібні "ручні" засоби контрольних точок, записані додатковими операторами `cout<<`.

Покрокове виконання програми. Переглядати процес виконання програми можна шляхом виконання операторів, записаних окремими рядками в тексті коду. Для цього використовують в середовищі програмування команди меню Debug -> Step Into (F11) або Debug -> Step Over (F10). Після кожного натискання на вказану клавішу система виконує один наступний рядок тексту програми (саме рядок цілий, а не окремий оператор) і чекає наступної команди. Отже, виконання програми зупиняється після кожного рядка операторів. Це надає змогу переглянути стан всіх величин, використаних в програмі. Наприклад, після низки кроків виконання операторів для фрагменту тексту програми можна бачити, наприклад, таке зображення (частина повних вікон):

```
15    }
16    // перегляд масиву і пошук від'ємних, якщо такі є
17    if(find)
18    {
19      for (int k = 0; k < N; k++)
20      {
21        if (num[k]<0 && num[k]>negative) negative = num[k];
22      } ≤2ms elapsed   num[k] 11
23      cout << "max among negative is " << negative << endl;
24    }
25    else { cout << "numbers negative not find.\n"; }
26    system("pause"); return 0;
27  }
```

100 %

Autos

Name	Value	Type
N	10	int
k	6	int
negative	-7	int
num	0x0043f7a8 {15, -9, 0, 32, -7, 22, 11, -4, -10, 20}	int[10]
num[k]	11	int

У вікні редактора тексту (верхнє) ліворуч від рядка номер 22 курсор жовтого кольору показує місце поточної зупинки. Сам рядок за номером 22 ще не виконаний на момент зупинки. Навівши курсор мишки на величину num[k], в спливаючому вікні видно значення 11 цієї величини на момент перегляду. Так само мишкою можна переглянути інші величини.

В режимі налагодження і покрокового виконання програми з'являється вікно Autos (нижнє). В ньому записані значення величин програми на момент зупинки і перегляду. Список величин змінюється автоматично залежно від позиції зупинки програми.

Блочне виконання програми. Покрокове виконання, показане вище, може бути занадто повільним способом візуалізації. Для швидшого перегляду можна визначити контрольні точки для зупинок виконання програми лише на окремих рядках коду. Визначають контрольні точки зупинок так: ставлять курсор позиції на потрібний рядок тексту програми і натискають F9 (команда Debug -> Toggle Breakpoint). Ліворуч від рядка з'явиться позначка червоного кольору.

Тепер, натискаючи клавішу F5 (команда Debug -> Start Debuggind / Continue), програма щоразу виконує наступний фрагмент до наступної контрольної точки і зупиняється. При цьому можна виконати перегляд величин так само, як показано вище.

Графічна візуалізація алгоритму

Для графічної візуалізації використовують малюнки, діаграми і анімації процедур виконання кожного оператора програми. Графічний спосіб спостереження вимагає спеціальної організації перетворення обчислених результатів до графічної форми, простіше кажучи, - до малюнка. Таке перетворення не може бути простим, а вимагає наявності окремих спеціалізованих графічних систем. Такі системи є, вони працюють автономно і незалежно від нашої робочої системи програмування.

Подаємо посилання на деякі відомі системи графічної візуалізації. Кожна система графічної візуалізації має окрему будову, їх не можна еквівалентно співставляти. Призначення кожної системи так само може бути різним.

https://algorithm-visualizer.org/

Інтерактивний візуалізатор алгоритму, виконаний дуже цікаво і якісно. Подає візуалізацію багатьох задач за різними розділами. Демонструє програмну реалізацію алгоритмів мовою JavaScript. Варто переглянути цей візуалізатор і обрати алгоритми, які є актуальними для навчання, розробки, вдосконалення.

https://thimbleby.gitlab.io/algorithm-wiki-site/

[Переходити на сайт треба через переглядач Chrome.]

Анімація інтерактивних алгоритмів Wiki. Алгоритми на цій вікі - це не просто анімації, вони побудовані на реальному інтерпретаторі, а візуалізація базується на фактичному коді Javascript, що працює. Код можна копіювати і використати для власних експериментів. Сайт має великий перелік доступних алгоритмів за багатьма темами.

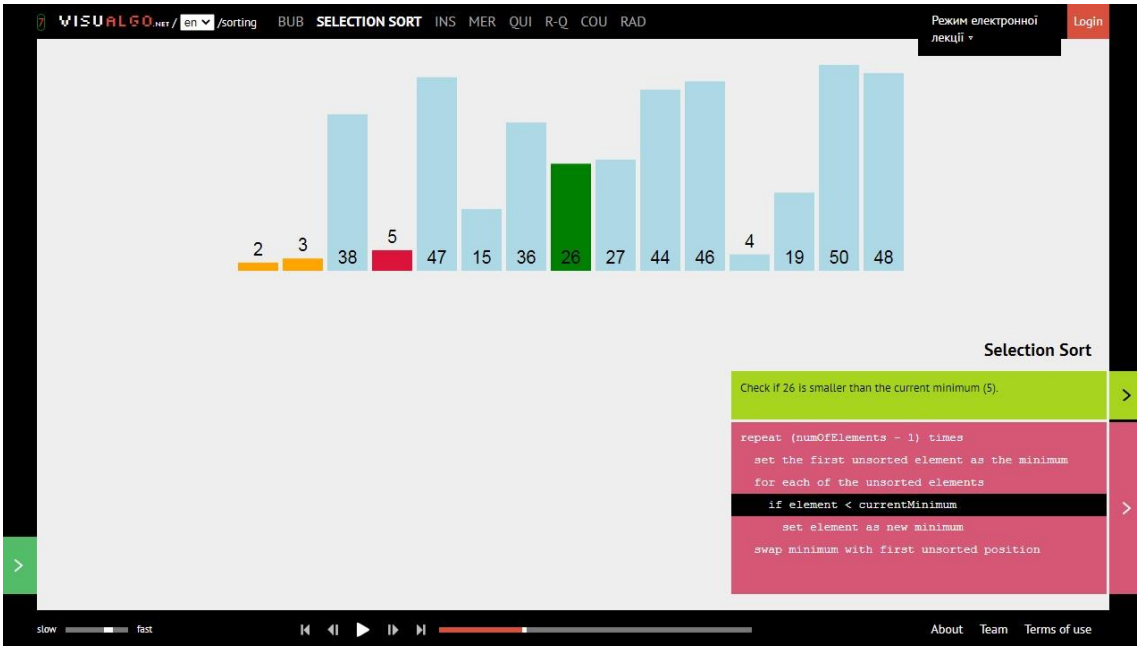
https://rosulek.github.io/vamonos/demos/

Динамічна візуалізація алгоритмів безпосередньо в браузері. Розглянуто перелік типових задач за розділами: сортування; рекурсивні обчислення; динамічне програмування; графи. Візуалізацію демонструють на конкретних прикладах. Приклади не складні і дозволяють отримати уявлення про динаміку виконання програми. Добре для початкового ознайомлення з поведінкою алгоритмів. Зразки алгоритмів демонструють кодом, близьким до поширених мов програмування.

https://visualgo.net/en

Візуалізація структур даних та алгоритмів за допомогою анімації. Цікава система візуалізації працює безпосередньо в браузері. Включає багато розділів прикладного програмування. Має окремо навчальні підсистеми для більшості розділів. Важливо, що дозволено самому будувати вхідні дані для алгоритмів, анімація буде виконана для своїх вхідних даних. Можна керувати режимами анімації: кроками/автоматично, повільніше/швидше. Система має вільний доступ в режимі дослідження.

Тут подано приклад кроку візуалізації алгоритму сортування вибором. Візуалізація виконана за допомогою системи https://visualgo.net/en, анотованої вище.



https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Візуалізації для структур даних та алгоритмів. Розробник - Університет комп'ютерних наук Сан-Франциско. Зручна для користування і якісно зроблена візуалізація за багатьма розділами програмування: основи (стек, черги, списки), рекурсія, індексування, методи сортування, структури даних типу купи, алгоритми на графах, динамічне програмування, геометричні алгоритми (дво- і три-вимірні матриці обертання, масштабування, зміни координат), інші.

https://www.101computing.net/data-visualisation-algorithms/

Сайт алгоритмів візуалізації даних. Можна шукати алгоритми для різних тем і задач. Подає приклади програмування для візуалізації різними алгоритмічними мовами. На сайті є також задачі і приклади для навчання.

https://regexr.com

RegExr - це онлайн-інструмент для вивчення, побудови і тестування регулярних виразів (RegEx/RegExp). Але використання поданих інструментів вимагає окремих експериментів.