https://github.com/KacperBudnik/AiSD

<ipython-input-1-1dd12c072ab6> in init (self, *items) self.list = [0]self.size = 0 8 ---> 9 self.add(*items) 10 11 def percUp(self,i): <ipython-input-1-1dd12c072ab6> in add(self, *items) elif isinstance(item, np.ndarray): 9.5 item=list(item) self.add(item) ---> 96 97 else: 98 self.insert(item) <ipython-input-1-1dd12c072ab6> in add(self, *items) for item in items: 87 if type(item) in (tuple, list): ---> 88 self.add(*item) elif type(item) in (dict, set): 89 self.add(*list(item)) <ipython-input-1-1dd12c072ab6> in add(self, *items) 96 self.add(item) 97 else: ---> 98 self.insert(item) 99 def _clear(self): # czyszczenie kopca (prywatna, by nie użyć przez przypadek) <ipython-input-1-1dd12c072ab6> in insert(self, k) self.list.append(k) 20 self.size = self.size + 1 21 ---> 22 self.percUp(self.size) 23 def findMin(self): <ipython-input-1-1dd12c072ab6> in percUp(self, i) 11 def percUp(self,i): while i // 2 > 0: 12 if self.list[i] < self.list[i // 2]:</pre> ---> 13 14 tmp = self.list[i // 2] 15 self.list[i // 2] = self.list[i] KeyboardInterrupt: In [49]: Tim Out[49]: array([0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 1.00040436e-03, 9.99927521e-04, 3.00073624e-03, 6.00147247e-03, 1.50039196e-02, 3.60085964e-02, 7.40170479e-02, 1.59030199e-01, 3.48716497e-01, 7.71439791e-01, 1.65170741e+00, 3.59301019e+00, 7.75503492e+00, 1.69674728e+01, 3.60844564e+01, 7.80425515e+01, 1.67152969e+02, 3.60247869e+02, 7.79478208e+02, 1.67020568e+03, 0.00000000e+00, 0.00000000e+00]) import matplotlib.pyplot as plt y=times[:26] x=range(26)Poniżej jest wykres czasu wykonania funkcji (na czerwono). Jeśli porównamy z wykresem funkcji $n \ln n$ (na niebiesko) to zauważymym, że obie funkcjie niemal dokładnie się pokrywają, nasza (czerwona) jest delikatni niżej, ponieważ w zależnośći od przypadku liczba wykonujemy mniej iteracji (tak jak można było się domyślać). Oczywiście oś niebieska i czerwona się różnią, ale różnica miedzy jedną a drugą to "stała czasowa" (na niebiesko ile zrobiliśmy iteracji, na czerwono jaki czas trwały. czerwony = niebieski * "stała"). fig, ax1 = plt.subplots() color = 'tab:red' ax1.set xlabel('\$2^n\$') ax1.set ylabel('real time', color=color) ax1.plot(x, y, color=color) ax1.tick params(axis='y', labelcolor=color) ax2 = ax1.twinx() # instantiate a second axes that shares the same x-axis color = 'tab:blue' $ax2.set_ylabel('$n\ln(n)$', color=color)$ # we already handled the x-label with ax1 ax2.plot(x, [2**i*np.log(2**i) for i in x], color=color) ax2.tick params(axis='y', labelcolor=color) fig.tight layout() # otherwise the right y-label is slightly clipped plt.show() 800 700 600 500 400 300 200 100 10 25 15 20 2" y=Tim[:-2] # dwa ostatnie nie dały rady się zrobić x=range(len(Tim[:-2])) fig, ax1 = plt.subplots() color = 'tab:red' ax1.set xlabel('\$2^n\$') ax1.set_ylabel('real time', color=color) ax1.plot(x, y, color=color) ax1.tick params(axis='y', labelcolor=color) ax2 = ax1.twinx() # instantiate a second axes that shares the same x-axis color = 'tab:blue' $ax2.set ylabel('$n\ln(n)$', color=color)$ # we already handled the x-label with ax1 ax2.plot(x, [2**i*np.log(2**i) for i in x], color=color)ax2.tick params(axis='y', labelcolor=color) fig.tight layout() # otherwise the right y-label is slightly clipped plt.show() 1750 -12 1500 -1.0 1250 -0.8 1000 -0.6 750 -0.4 500 0.2 - 0.0 Z wykresów możemy odczytać, że złożoność obliczeniowe jest rzędu $O(n \ln n)$ Zadanie 3 Jest to BinHeap z zmodyfikowanym jedynie konstruktorem oraz metodą insert import types import numpy as np class BinHeapN: def init (self,n:int,*items): # n - maksymalna wielkość kopca if type(n) is not int or not n >0: raise ValueError ("Maksymalna wielkość musi być liczbą naturalną (>0)") self.max size=n self.list = [0]self.size = 0self.add(*items) def percUp(self,i): while i // 2 > 0: if self.list[i] < self.list[i // 2]:</pre> tmp = self.list[i // 2]self.list[i // 2] = self.list[i] self.list[i] = tmp i = i // 2def insert(self,k): self.list.append(k) self.size = self.size + 1 if self.size<=self.max size: # jeśli wielkość jest mniejsza niż maksymalna self.percUp(self.size) if k>self.findMin(): # robi coś jeśli dodajemy element większy (mniejszy od razu jest usuwany) self.percUp(self.size) self.delMin() self.list.pop() # jeśli nowy jest najmniejszy do go usuwamy (dodany na końcu i jeszcze nie prze def findMin(self): if self.size==0: raise IndexError("Kopiec jest pusty") return self.list[1] def percDown(self,i): while (i * 2) <= self.size:</pre> mc = self.minChild(i) if self.list[i] > self.list[mc]: tmp = self.list[i] self.list[i] = self.list[mc] self.list[mc] = tmpi = mcdef minChild(self,i): **if** i * 2 + 1 > self.size: return i * 2 if self.list[i*2] < self.list[i*2+1]:</pre> return i * 2 else: **return** i * 2 + 1 def delMin(self): if self.size==0: raise IndexError("Kopiec jest pusty") retval = self.list[1] self.list[1] = self.list[self.size] self.size = self.size - 1 self.list.pop() self.percDown(1) return retval def buildHeap(self,alist): i = len(alist) // 2self.size = len(alist) self.list = [0] + alist[:] **while** (i > 0): self.percDown(i) i = i - 1def size(self): return self.size def isEmpty(self): return self.size == 0 def str (self): txt = "{}".format(self.list[1:]) return txt Własne "nie zmienione w stosunku do BinHeap" def repr (self): return str(self.list[1:]) def len (self): return self.size def add(self, *items): for item in items: if type(item) in (tuple, list): self.add(*item) elif type(item) in (dict, set): self.add(*list(item)) elif type(item) == types.GeneratorType: for i in item: self.add(i) elif isinstance(item, np.ndarray): item=list(item) self.add(item) else: self.insert(item) def clear(self): self.list=[0] self.size=0 def sort(self): sortedList=[] s=self.size while self.size>0: sortedList.append(self.delMin()) self.list=[0]+sortedList self.size=s In [216... a=BinHeapN(5,[2,3,2])Out[217... [2, 3, 2] In [218... a.add(12,12) In [219... Out[219... [2, 3, 2, 12, 12] a.add(3) Out[221... [2, 3, 3, 12, 12] a.add(5)Out[223... [3, 3, 5, 12, 12] In [224... a #niezmienione bo 2<3 Out[224... [3, 3, 5, 12, 12] Zadanie 4 Funkcja: derivate pobiera wyrażenie matematyczne, łączy wszystkie inne funkcjie i zwraca drzewo wyrażenia matematycznego (MathTree). derivate_to_latex działa podobnie jak poprzednia, ale pobiera od razu zmienną po której ma różniczkować i zwraca wynik • prepare zagnieżdża wyrażenia jedno w drugim ("wstawia nawiasy", ustala, że np. mnożenie robimy przed dodawaniem) correct decyduje o kolejności działań w obrębie działań równoważnych (np. dla $a \cdot b \cdot c$ ustala, które mnożenie pierwsze def derivate(txt): # zwraca drzewko matematyczne txt=txt.replace(" ", "") txt=txt.replace("**", "^") txt=txt.replace("\\", "/") txt=txt.replace(":", "/") txt=txt.replace("arc", "a") tree=MathTree(*prepare(txt)) return tree def derivate_to_latex(txt,variable): txt=txt.replace(" ", "") txt=txt.replace("**", "^") txt=txt.replace("\\", "/") txt=txt.replace(":", "/") txt=txt.replace("arc", "a") tree=MathTree(*prepare(txt)) return "\$\$"+tree.derivate_to_latex(variable)+"\$\$" # zwraca pochodną w latexu def correct(equation,lvl=4): # lvl - który operator rozpatrujemy (dzielenie przed mnożenim, odejmowanie przed dodawaniem) operators=["+","-","*","/","^"] if operators[lvl] in ("^",): reverse=True else: reverse=False for i in range(len(equation))[::(-1)**reverse]: if equation[i] == operators[lvl]: return correct(equation[:i-1]+[equation[i-1:i+2]]+equation[i+2:],lvl=lvl) **if** lvl>0: return correct(equation,lvl=lvl-1) return equation # Jeśli są działania równie ważne (np a*b*c), to wyznacza kolejność działania def prepare(txt): # Wyznacza kolejność działań itp. function=[[], ["ln"], ["log", "pow", "exp", "sin", "cos", "tan", "cot", "sec"], ["sqrt", "log2", "asin", "acos", "atan", "acot", "sinh", "cosh", "tanh", "coth", "sech"], ["log10"], ["cosech"] operators=["+","-","*","/","^"]#,"'"] unknown=False res=[] l=len(txt) while i<1: if txt[i]=="(": par=0 j=i while True: if txt[j]=="(": par**+=**1 if txt[j]==")": par**-=**1 **if** par==0: break res.append(prepare(txt[i+1:j])) unnown=True elif txt[i:i+6] in function[6]: j**=**i+6 if txt[j]=="(": par=0 while True: if txt[j]=="(": par**+=**1 if txt[j]==")": par**-=**1 **if** par==0: break res.append([txt[i:i+6],prepare(txt[i+6+1:j])]) else: res.append([txt[i:i+6],txt[i+6]]) unnown=True elif txt[i:i+5] in function[5]: #res.append(txt[i:i+5]) j=i+5 if txt[j]=="(": par=0 while True: if txt[j]=="(": par**+=**1 if txt[j]==")": par-=1 **if** par==0: break j**+=**1 res.append([txt[i:i+5],prepare(txt[i+5+1:j])]) res.append([txt[i:i+5],txt[i+5]]) unnown=True elif txt[i:i+4] in function[4]: j=i+4 if txt[j]=="(": par=0 while True: if txt[j]=="(": par**+=**1 if txt[j]==")": par-=1 **if** par==0: break res.append([txt[i:i+4],prepare(txt[i+4+1:j])]) else: res.append([txt[i:i+4],txt[i+4]]) unnown=True elif txt[i:i+3] in function[3]: #res.append(txt[i:i+3]) j**=**i+3 if txt[j]=="(": par=0 while True: if txt[j]=="(": par**+=**1 if txt[j]==")": par-=1 **if** par==0: break j**+=**1 res.append([txt[i:i+3],prepare(txt[i+3+1:j])]) else: res.append([txt[i:i+3],txt[i+3]]) unnown=True elif txt[i:i+2] in function[2]: #res.append(txt[i:i+2]) j=i+2 if txt[j]=="(": par=0 while True: if txt[j]=="(": par**+=**1 if txt[j]==")": par-=1 **if** par==0: break res.append([txt[i:i+2],prepare(txt[i+2+1:j])]) else: res.append([txt[i:i+2],txt[i+2]]) unnown=True elif txt[i] in operators: res.append(txt[i]) unknown=False else: if txt[i].isnumeric(): while txt[i:i+j+1].isnumeric() and j<1:</pre> if unknown: res.append('*') res.append(txt[i:i+j]) else: if unknown: res.append("*") res.append(txt[i]) unknown=True i +=1 res=correct(res) return res Classa MathTree, tu się dzieje cała magia. Pobiera wyrażenie matematyczne przygotowane przez funkcjie prepare. Takie wyrażenie jest to lista, która zawiera od 1 do 3 elementów. Jeśli zawiera 1, to jest to zmienna, stała, albo liczba, z którą nic więcej nie robimy. • Przy 2 elementach otrzymujemy funkcjię. Pierwszy element to sama funkcja, drugi natomiast to jej argument. Argument jest zapisywany w lewej gałęzi • Może zawierać jeszcze 3 elementy. Wtedy dostajemy działanie na dwóch wyrażeniach. Pierwszy i trzeci to owe wyrażenia, środkowy to działanie między nimi. (Przykładowo 2^3 daje ['2','^','3']) Classa ta zawiera dwie metody __repr__ to wyświetlajnia wyrażenia, oraz zwracania wyniku (w kodzie $L T_E X$) derivare_to_latex która zawiera całe mięsko i na niej się skupię Metoda ta pobiera zmienną po której chcemy różniczkować (jeśli jest ich za dużo, lub za mało zwraca błąd). Lista function zawiera listę obsługiwanych funkcji matematycznych, posegregowaną w listach wedługi długości ich nazw. Poniżej jest funkcja różniczkująca. Szuka operacji którą mamy wykonać i zwraca jej pochodną. Metoda może różniczkować szeroką klasę funkcji (przykładowo $f(x)^{g(x)}$), ale przezto wyniki nie zawsze są w najprostszej postaci (nie skraca ich). Przykładowo dla $f(x) = x^2$ funkcjia zwróci wynik, w którym każda działanie oraz liczba będzie w nawiasach, które po opuszczeniu wygląda następująco $f'(x) = x^2 \cdot \left(2 \cdot rac{1}{x} + \ln x \cdot 0
ight)$ Wynik jest poprawny, natomiast nie skrócony (no można się przyczepić co dla x <= 0). class MathTree: def init (self, equation): self.key=None self.left=None self.right=None if len(equation) == 1 or type(equation) == str: while type(equation) is list and len(equation) == 1: equation=equation[0] if type(equation) is list: if len(equation) == 2: self.key=equation[0] self.left=equation[1] else: self.key=equation[1] self.left=equation[0] self.right=equation[2] self.key=equation elif len(equation) == 2: self.key=equation[0] while type(equation[1]) is list and len(equation[1]) == 1: equation[1] = equation[1][0] self.left=MathTree(equation[1]) elif len(equation) == 3: self.key=equation[1] self.left=MathTree(equation[0]) self.right=MathTree(equation[2]) repr (self): if self.right==None: if self.left==None: return self.key return self.key+r"\left("+str(self.left)+r"\right)" else: return r"\left("+str(self.left)+r"\right)" + self.key +r"\left("+str(self.right)+r"\right)" def derivate to latex(self, variable): if len(variable)!=1 or variable.isnumeric(): raise ValueError("Błędna zmienna") function=[[], [], ["ln"], ["log", "pow", "exp", "sin", "cos", "tan", "cot", "sec", "csch"], ["sqrt","log2","asin","acos","atan","acot","sinh","cosh","tanh","coth","sech"], ["log10"], [] # ["cosech"] if self.key in ("+","-"): return self.left.derivate to latex(variable) + self.key + self.right.derivate to latex(variable) elif self.key=="*": return self.left.derivate to latex(variable)+r"\cdot "+str(self.right)+r"+"+str(self.left)+r"\cdot elif self.key=="/": return r"\dfrac{\left("+self.left.derivate to latex(variable)+r"\right)\cdot \left("+str(self elif self.key=="^": return r"\left("+str(self.left)+r"\right)^{\left("+str(self.right)+r"\right)}\cdot \left(\left("+st else: if self.key=="ln": return r"\dfrac{"+self.left.derivate to latex(variable)+r"}{"+str(self.left)+r"}" return r"\dfrac{"+self.left.derivate to latex(variable)+r"}{\left("+str(self.left)+r"\right)\cc elif self.key=="log2": return r"\dfrac{"+self.left.derivate to latex(variable)+r"}{\left("+str(self.left)+r"\right)\cq elif self.key=="exp": return r"e^{"+str(self.left)+r"}\cdot \left("+self.left.derivate to latex(variable)+r"\right)" elif self.key=="sin": return r"\cos\left("+str(self.left)+r"\right)\cdot \left("+self.left.derivate to latex(variable elif self.key=="cos": return r"-\sin\left("+str(self.left)+r"\right)\cdot \left("+self.left.derivate to latex(variab] elif self.key=="tan": return r"\dfrac{"+self.left.derivate to latex(variable)+r"}{\cos\left("+str(self.left)+r"\right elif self.key=="cot": return r"-\dfrac{"+self.left.derivate to latex(variable)+r"}{\sin\left("+str(self.left)+r"\right" elif self.key=="sec": return r"\sec\left("+str(self.left)+r"\right)\cdot \tan\left("+str(self.left)+r"\right)\cdot \1 elif self.key=="csc": return r"-\csc\left("+str(self.left)+r"\right)\cdot \cot\left("+str(self.left)+r"\right)\cdot \ elif self.key=="sqrt": return r"\dfrac{"+self.left.derivate to latex(variable)+r"}{\sqrt{}"+str(self.left)+r"}}" elif self.key=="asin": return r"\dfrac{"+self.left.derivate to latex(variable)+r"}{\sqrt{1-\left("+str(self.left)+r"\i elif self.key=="acos" return r"-\dfrac{"+self.left.derivate to latex(variable)+r"}{\sqrt{1-\left("+str(self.left)+r"} elif self.key=="": return r"\dfrac{"+self.left.derivate to latex(variable)+r"}{\left("+str(self.left)+r"\right)^2elif self.key=="": return r"-\dfrac{"+self.left.derivate to latex(variable)+r"}{\left("+str(self.left)+r"\right)^2 elif self.key=="sinh": return r"\cosh\left("+str(self.left)+r"\right)\cdot \left("+self.left.derivate to latex(variab) elif self.key=="cosh": return r"\sinh\left("+str(self.left)+r"\right)\cdot \left("+self.left.derivate to latex(variab) elif self.key=="tanh": return r"\sech\left("+str(self.left)+r"\right)^2\cdot \left("+self.left.derivate to latex(variation) elif self.key=="coth": return r"-\csch\left("+str(self.left)+r"\right)^2\cdot \left("+self.left.derivate to latex(vari elif self.key=="sech": return r"-\sech\left("+str(self.left)+r"\right)\cdot \tanh\left("+str(self.left)+r"\right)\cdot elif self.key=="": return r"-\sech\left("+str(self.left)+r"\right)\cdot \tanh\left("+str(self.left)+r"\right)\cdot elif self.key=="csch": return r"-\csch\left("+str(self.left)+r"\right)\cdot \coth\left("+str(self.left)+r"\right)\cdot elif self.key=="": return elif self.key=="": return if self.key==variable: return r"1" else: return r"0" W przykładach kożystałem z rozszerzenia jupytera Python Markdown , które pozwala uzywać zmiennych jupytera (pythona) w markdown'ie. Przykładowo a=derivate to latex("x^2", "x") print(a) $\$ \left(x\right)^{\left(2\right)}\cdot \left(2\right)\cdot \dfrac{\left(1\right)}{\left(x\right)}+\ln\left(2\right)} $t(x\left(0\right)\right) \$ Zamiast przekopiować powyższy kod wystarczy dać { {a} } (użyłem teraz spacji by rozszerzenie nie działało), by otrzymać poniższy wynik {{a}}} Uwaga! By dane rozszerzenie działało notatnik musi być zaufany, oraz niestety nie ma możliwości wyeksportować pliku z poniższymi wynikami. Przykłądy innych pochodnych a=derivate_to_latex("x^x","x") {{a}}} a=derivate to latex(" $cos(x)^(sin(x)+x^2)$ ","x") {{a}}} In [34]: a=derivate to latex("ax^2+bx+c-12","x") Działą z parametrami {{a}} a=derivate to latex("ax^2+bx+c-12","a") Nieważne jakie wybierzemy (przykładowo a) {{a}} a=derivate to latex("ax^2+bx+c-12","1") ValueError Traceback (most recent call last) <ipython-input-31-f065a5f7f65e> in <module> ----> 1 a=derivate_to_latex("ax^2+bx+c-12","1") <ipython-input-2-82cbc111d9ed> in derivate to latex(txt, variable) 17 18 tree=MathTree(*prepare(txt)) ---> 19 return "\$\$"+tree.derivate_to_latex(variable)+"\$\$" # zwraca pochodna w latexu 21 **def** correct (equation, lvl=4): <ipython-input-29-252a757a72b6> in derivate to latex(self, variable) def derivate to latex(self,variable): 44 if len(variable)!=1 or variable.isnumeric(): ---> 45 raise ValueError("Błędna zmienna") 46 function=[ValueError: Błędna zmienna Po liczbie nie działa Może bardziej skomplikowany przykład a=derivate to latex("23x","x") Można ominąć znaki mnożeni {{a}} a=derivate to latex(" $\sin(x)$ ^ $\cos(x)$ ^x",'x') {{a}}

KeyboardInterrupt

---> 11

12 13

<ipython-input-48-66a24169f6f1> in <module>

a=BinHeap(numbers[:2**i])

10 for i in range(n+1):

a.sort()

T=time.time()

Traceback (most recent call last)