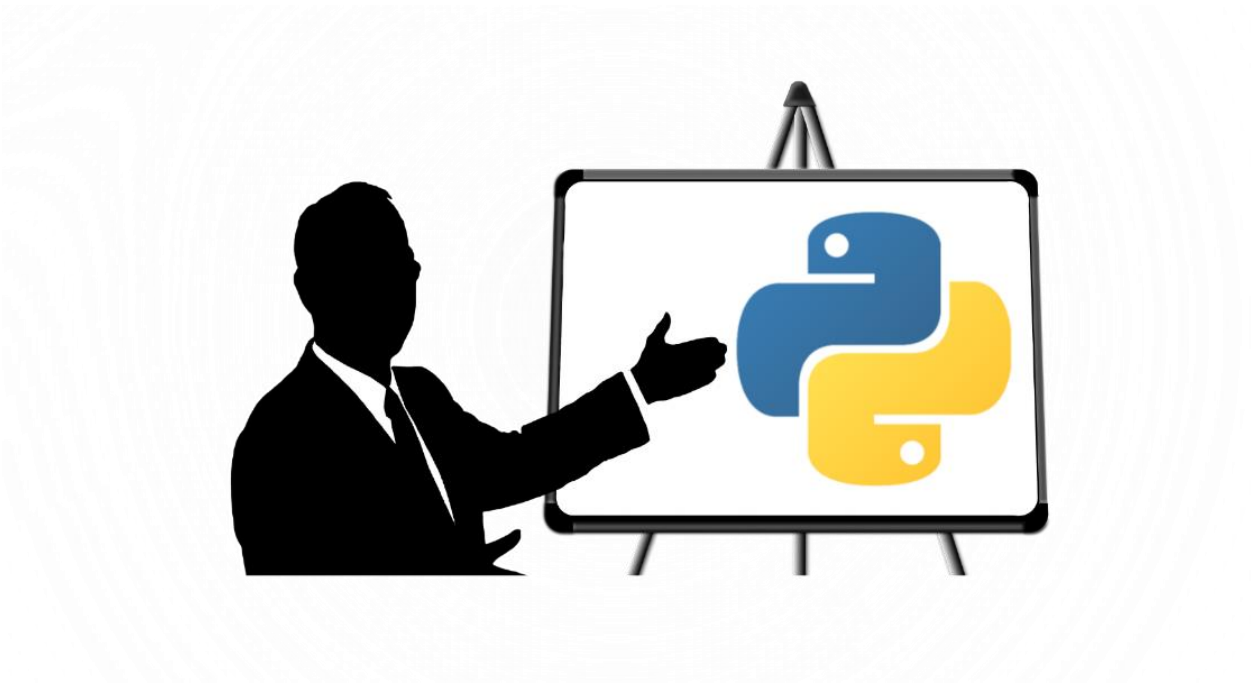


Myślenie algorytmiczne - Struktury danych i algorytmy, które musisz znać

Podręcznik kursowy

W tym miejscu zwykle pojawia się informacja o tym kto i jak może posługiwać się tym podręcznikiem. Bez regułek prawnych odwołam się po prostu do kilku prostych zasad, które oddają sens kto, kiedy i jak może wg zamysłu autora korzystać z tego materiału:

- Ten podręcznik jest integralnym elementem kursu online.
- Możesz z niego korzystać będąc uczestnikiem tego kursu. Podręcznik jest dla Ciebie i korzystaj z niego do woli – drukuj, wypełniaj, uzupełniaj, przeglądaj, póki Twoim celem jest samodzielne opanowanie tematu.
- Proszę nie umieszczaj go w publicznie dostępnych miejscach, jak blogi, repozytoria git hub, chomik itp.
- Nie wykorzystuj go w innych celach, np. organizacji własnych szkoleń, gdzie występujesz np. jako instruktor.
- Jeśli nie posłuchasz moich próśb, to jako autor zapiszę się do ZAIKS-u i następnym razem kupując smartfona zapłacisz za niego kilkaset złotych więcej 😜, więc może lepiej po prostu przestrzegaj praw autorskich 😊

Z góry dziękuję!

Rafał Mobilo © 2022

Zapraszam do odwiedzenia strony:

http://www.kursyonline24.eu/

Review 2023-12-02

Spis treści

Ta strona celowo jest pusta

Wprowadzenie – o kursie

Od znajomości słówek, do opanowania języka obcego jest długa droga. Podobnie jest z programowaniem. Znajomość instrukcji języka, a znajomość algorytmów to dwie różne rzeczy.

Ale właściwie, dlaczego wmawia się nam, że trzeba znać algorytmy? Komputery są coraz szybsze, pamięci mamy coraz więcej, a programiści mają do dyspozycji cały arsenał gotowych funkcji i często wcale nie muszą pisać własnych implementacji złożonych obliczeń.

Tak? To powiedz to w Google, Microsoft, IBM albo w innej firmie, która na poważnie podchodzi do tworzenia oprogramowania, sama tworzy innowacyjne rozwiązania, sztuczną inteligencję i narzędzia do pracy z Big Data. Dla takich firm tworzenie nowych algorytmów to konkretne pieniądze zaoszczędzone na mocy obliczeniowej, to otwarcie zupełnie nowych możliwości biznesowych, które bez odpowiednich algorytmów byłyby poza ich zasięgiem. Tam nie tylko tworzy się algorytmy, ale jeszcze je patentuje! Takie firmy zatrudniając programistów, chcą, aby znali oni algorytmikę, wiedzieli na jakie pułapki trzeba być uczulonym, a w jakich przypadkach można skorzystać z już gotowych rozwiązań. Nie dziwi więc, że od kandydatów oczekuje się rozwiązania zagadek programistycznych, czasami nawet na kilka sposobów i porównanie, który z nich i dlaczego jest lepszy.

Znając algorytmy – to chyba najważniejsze – poradzisz sobie z nietrywialnymi problemami, napiszesz efektywniejszy kod, który zadziała nawet na słabszym sprzęcie. Modyfikując algorytmy stajesz się naukowcem, rozbudujesz swoje portfolio i zwiększasz szanse na lepszą pracę. Na dodatek algorytmy się nie starzeją. Pojawi się nowa generacja komputerów, nowe języki programowania, ale algorytmy będą te same. No i wreszcie, argument nie do zbicia – algorytmy są po prostu fajne. To taka forma rozrywki umysłowej dla zaawansowanych. Nie krzyżówka, nie sudoku, ale myślenie algorytmiczne. Steve Jobs, chociaż nie jest dla mnie idolem, powiedział kiedyś „Każdy powinien umieć programować, bo to uczy myślenia”. Z tym zdaniem się zgadzam.

Ten kurs jest przeznaczony dla tych, co lubią pomyśleć, na poważnie biorą się za programowanie, znają już podstawy Pythona. Znajdziesz tu opis około dwudziestu algorytmów i powiązanych z nimi struktur danych. Zwykle zaczynamy od omówienia teorii stojącej za danym algorytmem, a potem przechodzimy do implementacji w Pythonie. Do kursu jest dołączony PDF z notatkami do każdej lekcji i zadaniami do samodzielnego oprogramowania oraz z rozwiązaniami tych zadań.

Kurs jest dynamiczny, może nawet czasami za bardzo, ale gdy korzysta się z e-learningu, nie powinien to być problem. Można zatrzymać, przewinąć, spowolnić lub przyśpieszyć.

Chcesz zająć się programowaniem na poważnie – to musisz znać algorytmy. Zapraszam na kurs!

Miłej nauki!

Rafał

Jak się uczyć?

Skoro tutaj zaglądasz, to znaczy, że planujesz poznać algorytmy. To świetnie!

Naukę oczywiście zorganizujesz sobie po swojemu, ale pozwól, że zaproponuję kilka sposobów nauki, a Ty sam/a wybierzesz, co z tego Ci się podoba, a co wolisz zrobić po swojemu

1. **Co za dużo to niezdrowo** – nie rób na raz za dużo materiału. Nie od razu Kraków zbudowano. Jedna lub dwie lekcje na dzień powinny wystarczyć.
2. **Liczy się regularność** – niekoniecznie uczyć trzeba się codziennie, ale jeśli postanowisz przerabiać lekcje we wtorki, czwartki i soboty to już coś!
3. **Wykonuj zadania praktyczne**. Od samego oglądania filmów się nie nauczysz. Trzeba samodzielnie rozwiązywać problemy, które na pewno się pojawią!
4. **Zmieniaj treść poleceń na własną rękę**. Wykonaj podobny przykład na innych danych. Im więcej kreatywności podczas nauki, tym więcej się zapamiętuje
5. Uczę się uczę, a do głowy nie wchodzi – wszyscy tak mamy i to pewnie dlatego nauka w szkole trwa aż tyle lat! **Od czasu do czasu zrób sobie powtórkę**. Przecież nikt Cię nie goni i nie rozlicza z postępów.
6. Wracaj do zadań i rozwiązuj je wielokrotnie. Jeśli potrafisz je rozwiązać – świetnie przerabiaj następną lekcję.
7. Jeśli zadania sprawiają problem, wróć do notatki lub lekcji – zobaczysz, że słuchając drugi raz tego samego, materiał nie będzie już taki trudny
8. Notatki w podręczniku są dla Twojej wygody. Niestety wygoda leży blisko lenistwa. Nie bądź leniem. Przygotuj sobie zeszyt lub kilka luźnych kartek i **zapisuj to czego się uczysz**. To co wejdzie oczami lub uszami, będzie wychodzić rękami i… nie ma wyjścia – po drodze zahaczy o mózg 😊
9. Jeśli możesz – wydrukuj sobie podręcznik, dopisuj do niego własne notatki, uwagi itp.
10. Kiedy osiągniesz jakiś „kamień milowy”, ukończysz sekcję kursu, a może nawet cały kurs – **daj sobie nagrodę** – to niesamowicie zwiększa motywację!
11. **Nie bój się korzystać z innych materiałów**: książek, blogów, forów itp. Część z nich na początku może być nieco za trudna, ale nic nie stoi na przeszkodzie, żeby na początku nic nie mówić, tylko słuchać 😊.
12. Kiedy już ukończysz kurs – **zaktualizuj CV na LinkedIn**, pochwal się swoim certyfikatem, daj się odnaleźć rekrutrom, zaproś mnie do znajomych (link w profilu). Chętnie potwierdzę Twoją nową umiejętność!

    Powodzenia!

    Rafał

Listy list, słowniki słowników?

Notatka

Popularne sposoby zapisywania informacji o obiektach w Pythonie to:

- Instancje klasy
- Słowniki

Kiedy pracujesz z większą liczbą takich obiektów możesz je umieszczać w postaci listy tych obiektów. Np. zakładając, że ben, jan i ann to słowniki, to możesz je umieścić w jednej liście następującym poleceniem:

```
contact_list = [ben, jan, ann]
```

Niestety wyszukiwanie obiektów w liście nieposortowanej wiąże się z koniecznością przejrzenia jej elementów, jeden po drugim, aż zostanie znaleziony ten właściwy element. Można wyeliminować ten problem przechowując dane w słowniku, w którym:

- Kluczem jest nazwa identyfikująca dany obiekt
- Wartością jest sam obiekt

Przepisanie listy słowników do postaci słownika słowników można dokonać za pomocą następującego kodu:

```
contact_dict = {}

for c in contact_list:
    contact_dict[c["name"]] = c
```

Aby odwołać się do danych z jednej wybranej pozycji (tutaj Ann) można posłużyć się poleceniami:

```
print(contact_dict["Ann"])
print(contact_dict["Ann"]["email"])
```

Lab

1. Student ma swoim planie zajęć na dzisiaj o godzinie 9:00 wykład z Mikrobiologii prowadzony przez profesora Wirusa, o 12 ćwiczenia z Chemii z doktorem Kolbą oraz o 14 wykład z Etyki z doktorem Ojej.
2. Zapisz informacje o planie dnia dla tego studenta w postaci:
   - Listy słowników
   - Słownika słowników – jako klucza użyj nazwy przedmiotu
3. Napisz pętlę, która z tych obiektów wyświetli tylko godzinę rozpoczęcia zajęć i temat (z pominięciem typu zajęć i prowadzącego)
   Uwaga: Iterując po słowniku możesz skorzystać z metody items(). W poniższym przykładzie k przyjmuje wartość klucza, a v jest wartością przechowywaną w słowniku:

```
for k,v in plan_dict.items():
```

Propozycja rozwiązania – pobierz kod

```
a1 = {
    "hour": "9:00",
    "type": "lecture",
    "topic": "Microbiology",
    "teacher": "prof Virus"
}

a2 = {
    "hour": "12:00",
    "type": "lab",
    "topic": "Chemistry",
    "teacher": "prof Kolba"
}

a3 = {
    "hour": "14:00",
    "type": "lecture",
    "topic": "Ethics",
    "teacher": "prof Ojej"
}

plan_list = [a1, a2, a3]

plan_dict = {}
for position in plan_list:
    plan_dict[position["topic"]] = position


for position in plan_list:
    print(f"{position['topic']} at {position['hour']}")


for k,v in plan_dict.items():
    print(f"{v['topic']} at {v['hour']}")
```

Przetwarzanie danych z list. Funkcja lambda

Notatka

Dane w listach można łatwo przetworzyć na kilka metod:

- Generowanie wartości nowej listy poprzez wywołanie funkcji

```
[f(x) for x in my_list]
```

- Generowanie wartości nowej listy poprzez funkcję anonimową lambda

```
[(lambda x: x*x)(x) for x in my_list]
```

- Generowanie wartości nowej listy przez generator

```
[x*x for x in my_list]
```

To, którą z tych metod wybierzesz najbardziej zależy od czynności, jaka ma być wykonana na liście. Jeśli ta czynność jest prosta rozważ korzystanie z generatora lub funkcji lambda. Jeśli do wykonania masz więcej złożonych czynności pomyśl o stworzeniu dedykowanej funkcji.

Lab

1. W programie przetwarzasz listę zawierającą imię i nazwisko:

```
names = ['John Johnson', 'Alicja Policja', 'Wladimir Wladymirowicz']
```

2. Napisz funkcję get_sub_name(name, part), która:
   a. Przekazaną nazwę rozbije na dwie części dzieląc ją ze względu na spację (można użyć metody **split** wywoływanej na rzecz napisu)
   b. Zwróci część określoną przez **part** – jeśli **part** == 0, to zwrócone będzie imię, a jeśli **part** == 1 to zwrócone będzie nazwisko.
3. Korzystając z listy **names** oraz funkcji **get_sub_name** utwórz listę zawierającą tylko imiona i drugą listę zawierającą tylko nazwiska.
4. A teraz zrób to samo, ale skorzystaj z funkcji anonimowej
5. A teraz zrób to samo, ale skorzystaj z generatora

Propozycja rozwiązania – pobierz kod

```
names = ['John Johnson', 'Alicja Policja', 'Wlodimir Wladymirowicz']

def get_sub_name(name, part):
# name that should be splitted by space
# part - which part of the name should be returned (0 - FirstName, 1 - LastName)
    first_name = name.split(' ')[part]
    return first_name


first_names = [get_sub_name(name, 0) for name in names]
last_names = [get_sub_name(name, 1) for name in names]
print(first_names)
print(last_names)


first_names = [(lambda name: name.split(' ')[0])(name) for name in names]
last_names = [(lambda name: name.split(' ')[1])(name) for name in names]
print(first_names)
print(last_names)



first_names = [name.split(' ')[0] for name in names]
last_names = [name.split(' ')[1] for name in names]
print(first_names)
print(last_names)
```

Funkcje pracujące na listach

Notatka

Lista to przykład iteratora. Lista ma „ukrytą” funkcję __iter__() zwracającą iterator. Ten iterator z kolei posiada metodę __next__(), która zwraca kolejne wartości. Obiekty „iterable” czyli posiadające iterator można iterować 😊 czyli przetwarzać je w pętli. Tego typu obiekty można łatwo konwertować do postaci listy korzystając z polecenia list(….).

Przydatne funkcje w pracy z listami to:

- map – uruchamia wskazaną funkcję, dla każdego elementu listy

```
list(map(lambda x: x.upper(), ['cat', 'dog', 'cow']))
```

- filter – filtruje listę. Zwracane są te elementy, dla których funkcja zwraca True

```
list(filter(lambda x: 'o' in x, ['cat', 'dog', 'cow']))
```

- reduce – wyznacza dla listy jedną wartość w oparciu o funkcję uruchamianą na parach wartości z listy

```
from functools import reduce
reduce(lambda x,y: x if x>y else y, [11, 22, 33, 1])
```

Funkcje sorted, nlargest i nsmallest (oraz wiele innych) pozwalają na przekazanie przez argument key funkcji, której zadaniem jest wskazanie wartości, po których ma się odbywać sortowanie lub wybieranie największych lub najmniejszych wartości.

```
ids = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']
sorted(ids, key=lambda x: int(x[2:]))
nlargest(3, ids, key=lambda x: int(x[2:]))
nsmallest(3, ids, key=lambda x: int(x[2:]))
```

Lab

1. Posortuj listę [1,45,672,7265,16] ze względu na ostatnią cyfrę (w tym przypadku np. 16>672, bo ostatnia cyfra w 16 to 6 i 6 jest większe od 2, która jest ostatnią cyfrą w 672)
2. Korzystając z funkcji reduce wyznacz długość najdłuższego napisu w liście:

```
codes = ['JPID', 'JJJPPP', 'XXX', 'JDU']
```

3. Dane są listy:

```
capitals =['Rome', 'Paris', 'Madrid']
cities = ['Rome', 'Napoli', 'Rimini', 'Paris', 'Barcelona', 'Madrid', 'Marceille']
```

   Korzystając z funkcji filter, wyświetl te miasta z cities, które nie są stolicami (nie występują na liście capitals)

Propozycja rozwiązania – pobierz kod

```
numbers = [1,45,672,7265,16]
sorted_numbers = sorted(numbers, key=lambda x: x % 10)
print(sorted_numbers)


from functools import reduce

codes = ['JPID', 'JJJPPP', 'XXX', 'JDU']
print(reduce(lambda a, b:  a if len(a) > len(b) else b, codes))


capitals =['Rome', 'Paris', 'Madrid']
cities = ['Rome', 'Napoli', 'Rimini', 'Paris', 'Barcelona', 'Madrid', 'Marceille']

print(list(filter(lambda c: c not in capitals, cities)))
```

Listy, słowniki, zbiory, tuple – kiedy z czego korzystać?

Notatka

Najważniejsze struktury danych, natywnie dostępne w Pythonie to:

List

- Mogą być generowane przez funkcję **range(start, stop, increment)**. Zwracany obiekt jest typu **range**, ale można go skonwertować do listy np. **list(range(20,10,-2))**
- Większość operacji wykonywanych na listach ma złożoność obliczeniową O(N)

Tuple

- Tuple jest podobne do listy, ale jest niezmienne (immutable), jeśli w oparciu o tuple chcesz utworzyć nowy, z dodatkowymi wartościami to można użyć „sztuczek”

```
(*my_tupple, 99)
(my_tupple + (99,))
```

- Złożoność jest taka, jak w listach, czyli w większości operacji O(N)

Set

- Zbiór wartości, bez porządku deklarowany przez **{}**, np. **drinks = {‘milk’, ‘tee’},** często wykorzystywane operatory to in (czy element należy do zbioru) | (suma) & (część wspólna)
- Praca z pojedynczymi elementami zbioru ma złożoność O(1)

Dictionary

- Iterując po słowniku często korzysta się z metody items()

```
for k,v in my_dictionary.items():
  print(k, v)
```

- Dzięki mechanizmowi wyznaczanych przez Pythona wartości hash, większość operacji ma złożoność O(1)

Lab

1. Dodaj stronę https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt do ulubionych (plik dostępny też w materiałach kursowych)
2. Porównamy wydajność list i set pod względem wyszukiwania danych operatorem in. Na moim komputerze program wykonywał się ok. 15 sek. Zmodyfikuj max_limit aby wydłużyć/skrócić czas trwania. Zaimportuj moduł time i zainicjuj zmienne:

```
import time
max_limit = 50000
my_list = list(range(max_limit))
my_set = set(range(max_limit))
```

3. Napisz pętlę, która dla każdej liczby p z zakresu od 0 do max_limit sprawdzi, czy ta liczba znajduje się w liście my_list. Napisz drugą pętlę sprawdzającą to samo w zbiorze my_set
4. Przed rozpoczęciem pętli i po jej zakończeniu dodaj instrukcje pobierające aktualny czas. Dodaj komunikat wyświetlający różnicę czasu. Ustal, co działa szybciej: list czy set?

```
start = time.time()
# tutaj umieść pętlę
stop = time.time()
print(f'Operation duration was {stop - start}')
```

Propozycja rozwiązania – pobierz kod

```
import time


max_limit = 50000
my_list = list(range(max_limit))
my_set = set(range(max_limit))



# checking time for list search

start = time.time()

for p in range(max_limit):
    is_present = p in my_list

stop = time.time()

print(f'List search operation duration was {stop - start}')



# checking time for set search

start = time.time()

for p in range(max_limit):
    is_present = p in my_set

stop = time.time()

print(f'Set  search operation duration was {stop - start}')
```

Listy dwukierunkowe

Notatka

Lista dwukierunkowa to kolekcja obiektów ułożonych „jeden za drugim”, z których każdy zna adres wyłącznie swoich sąsiadów „przed” i „po”. Taką listę można w wygodny sposób przechodzić w pętli while. Warunek zakończenia pętli będzie sprawdzał, czy osiągnięto już ostatni element listy.
Oto przykładowy słownik, który mógłby być elementem takiej dwukierunkowej listy:

```
wi1 = {
    "name" : "A Standup Cloud Team",
    "start_hour" : 8,
    "duration_min" : 15,
    "prev" : None,
    "next" : None
}
```

Pętla przetwarzająca wszystkie elementy z listy może mieć taki kod:

```
item = work
while item:
    print(item["name"])
    item = item["next"]
```

Dodawanie nowego elementu do listy na początku tej listy może realizować funkcja:

```
def add_item_begin(my_collection, new_item):
    new_item["next"] = my_collection
    new_item["prev"] = None

    if my_collection:
        my_collection["prev"] = new_item

    my_collection = new_item
    return my_collection
```

Dodawanie nowego elementu na końcu listy wykonuje kolejna funkcja:

```
def add_item_end(my_collection, new_item):

    if my_collection == None:
        my_collection = new_item
        new_item["next"] = None
        new_item["prev"] = None
        return my_collection
    else:
        item = my_collection
        while item["next"]:
            item = item["next"]

        item["next"] = new_item
        new_item["next"] = None
        new_item["prev"] = item
        return my_collection
```

Lab

1. Napisz funkcje, które odpowiadają za usunięcie pierwszego del_item_begin(my_collection) lub ostatniego elementu del_item_end(my_collection) z listy dwukierunkowej. Na kolejnej stronie znajdziesz kod z lekcji, Twoim zadaniem jest tylko dopisanie nowych funkcji i ich przetestowanie.

Propozycja rozwiązania – pobierz kod

```
wi1 = {
    "name" : "A Standup Cloud Team",
    "start_hour" : 8,
    "prev" : None,
    "next" : None
}
wi2 = {
    "name" : "B Architecture Board",
    "start_hour" : 9,
    "prev" : None,
    "next" : None
}
wi3 = {
    "name" : "C Cloud Training",
    "start_hour" : 12,
    "prev" : None,
    "next" : None
}

wi1["next"] = wi2
wi2["prev"] = wi1
wi2["next"] = wi3
wi3["prev"] = wi2
work = wi1

def add_item_begin(my_collection, new_item):
    new_item["next"] = my_collection
    new_item["prev"] = None

    if my_collection:
        my_collection["prev"] = new_item

    my_collection = new_item
    return my_collection

wi4 = {
    "name" : "___ Manager meeting",
    "start_hour" : 7,
    "prev" : None,
    "next" : None
}
work = add_item_begin(work, wi4)

def add_item_end(my_collection, new_item):

    if my_collection == None:
        my_collection = new_item
        new_item["next"] = None
        new_item["prev"] = None
        return my_collection
    else:
        item = my_collection
        while item["next"]:
            item = item["next"]

        item["next"] = new_item
        new_item["next"] = None
        new_item["prev"] = item
        return my_collection

wi5 = {
    "name" : "D Compliance Check",
    "start_hour" : 15,
    "prev" : None,
    "next" : None
}
```

```
work = add_item_end(work, wi5)

item = work
while item:
    print(item["name"])
    item = item["next"]

################################################################
# HERE STARTS SOLUTION OF THE LAB, TRY TO DEVELOP IT YOURSELF #
################################################################


def del_item_begin(my_collection):
    # item is the element that should be removed
    item = my_collection
    # execute only if the collection was not empty
    if item:
        # my collection should point to the next element
        my_collection = item['next']
        # if the collection is not empty, it's first element property 'prev' should
point to None
        if my_collection:
            my_collection['prev'] = None

    return my_collection


work = del_item_begin(work)

print('--- After deletion of the first element ---')
item = work
while item:
    print(item["name"])
    item = item["next"]


def del_item_end(my_collection):
    # if the collection is not empty
    if my_collection:
        # go to the last element
        item = my_collection
        while item["next"]:
            item = item["next"]

        # the last element will be removed and this will be a new last element
        new_last_item = item['prev']
        # if such new last element exists, we need to assign to the prev property
value of None
        if new_last_item:
            new_last_item['next'] = None
        else:
            # but if the new last element is None, it means that we are removing the
last element from the list
            # so now, the collection is empty
            my_collection = None

    return my_collection


work = del_item_end(work)

print('--- After deletion of the last element ---')
item = work
while item:
    print(item["name"])
    item = item["next"]
```

Ta strona celowo jest pusta

Kolejka i stos – FIFO i LIFO

Notatka

Kolejka FIFO (first in first out) to kolekcja, do której można dodawać obiekty. Podczas pobierania obiektów z kolejki, wcześniej zostaną zwrócone te obiekty, które były dodawane do kolejki na samym początku. Odpowiada to pojęciu kolejki w intuicyjny sposób.

Stos LIFO (last in first out) to kolekcja, do której można dodawać obiekty. Podczas pobierania obiektów z kolejki, wcześniej zostaną zwrócone te obiekty, które były dodawane do kolejki na samym końcu. Odpowiada to np. intuicyjnemu pojęciu stosu kartek na biurku.

Struktura pamięciowa wykorzystywana przez FIFO i LIFO może być taka sama i może to być np. lista dwukierunkowa. Co jest istotne, to jednak dodawanie i pobieranie elementów z takiej kolekcji we właściwy sposób, odpowiadający kolejce lub stosowi.

Lab

1. W tym laboratorium zaimplementujesz kolejkę i stos wykorzystujące jako reprezentację pamięciową listę.
2. Kolejka – FIFO
    a. Utwórz pustą listę list_fifo
    b. Dodaj do tej listy 3 rzeczy, które powinien zrobić w swoim życiu mężczyzna: „Plant a tree”, „Build a house”, „Have a son”
    c. Napisz pętlę while, która pobierze te teksty w kolejności ich dodawania. Wskazówka: posłuż się metodą pop z odpowiednim argumentem https://docs.python.org/3/tutorial/datastructures.html
3. Stos – LIFO – do ciężarówki zapakowano pewne pakunki. Operator wózka widłowego zrobił to w pewnej specyficznej kolejności, oto ona: „Blue Box with Glass”, „Bag with Presents”, „Barrel of Beer”, „Cage with a Tiger”. Pakunki powinny być wypakowane z ciężarówki w odwrotnej kolejności, dzięki temu żadna paczka nie spadnie i się nie uszkodzi.
4. Utwórz pustą listę list_lifo
5. Dodaj do niej listę pakunków w kolejności zgodnej z opisaną powyżej
6. Napisz pętlę while, która ustali w jakiej kolejności należy wypakowywać paczki

Propozycja rozwiązania – pobierz kod

```
list_fifo = []

list_fifo.append("Plant a tree")
list_fifo.append("Build a house")
list_fifo.append("Have a son")

print(list_fifo)

while len(list_fifo) > 0:
    task = list_fifo.pop(0)
    print(task)


list_lifo = []

list_lifo.append('Blue Box with Glass')
list_lifo.append('Bag with Presents')
list_lifo.append('Barrel of Beer')
list_lifo.append('Cage with a Tiger')

print(list_lifo)

while len(list_lifo) > 0:
    task = list_lifo.pop()
    print(task)
```

Moduł queue

Notatka

Ponieważ kolejkowanie obiektów jest często wykonywaną czynnością w programach, powstały specjalne klasy, które pozwalają budować kolejkę lub stos. Jednym z takich modułów jest **queue**. Moduł trzeba na początku zaimportować:

```
import queue
```

Następnie do dyspozycji są 3 klasy:

- Queue – odpowiadająca za kolejkę FIFO:

```
q = queue.Queue()
```

- LifoQueue – odpowiadająca za stos LIFO:

```
q = queue.LifoQueue()
```

- PriorityQueue – odpowiadającą za kolejkę FIFO z priorytetami:

```
q = queue.PriorityQueue()
```

Dodawanie elementów wykonuje się metodą **put()**, a pobranie metodą **get()**. Aby sprawdzić, ile obiektów jest w kolejce można posłużyć się metodą **qsize()**. W przypadku Queue i LifoQueue metoda **put()** przyjmuje jeden argument (obiekt do umieszczenia w kolejce), ale dla PriorityQueue należy przekazać dwa argumenty: priorytet i obiekt. Metoda **get()** w przypadku PriorityQueue w pierwszej kolejności zwraca obiekty z wyższym priorytetem.

Lab

1. Dana jest ścieżka my_path='/home/boss/data/projects/bakery/prices.csv'
2. Ponieważ właśnie poznaliśmy moduł queue… wyświetl poszczególne elementy tej ścieżki w odwrotnej kolejności:
    a. Pętlą for wpisz do odpowiedniej klasy z modułu queue wynik podziału ścieżki ze względu na znak slash (my_path.split('/')
    b. Pętlą while wyświetl elementy z kolejki – najpierw powinno być wyświetlone prices.csv, potem bakery, potem projects itd.
3. Została Ci może minutka? To zapoznaj się z zastosowaniem kolejki z priorytetem: https://youtu.be/3aqc0HagEMc (Film jest dość stary ale….)

Propozycja rozwiązania – pobierz kod

```
import queue


my_path = '/home/boss/data/projects/bakery/prices.csv'

q = queue.LifoQueue()

for part in my_path.split('/'):
    q.put(part)

while q.qsize()>0:
    print(q.get())
```

PANDAS i DataFrame

Notatka

Do analizy dużej ilości danych przydają się wyspecjalizowane moduły, np. PANDAS. Zainstalujesz go poleceniem:

```
pip install pandas
```

Głównym obiektem służącym do przechowywania danych jest DataFrame i można go inicjować tak:

```
df = pd.DataFrame(
    [
        [1, 'Math for Juniors', 3, False],
        [2, 'Geometry in Nature', 1, False]
    ]
)
df.columns = ['Id', 'Name', 'Amount', 'Promo']
```

Wyświetlając dane można je filtrować po kolumnach lub wierszach:

```
df[['Name', 'Amount']]      # argumentem jest lista z nazwami kolumn
df.iloc[1:,1]               # pierwszy argument określa wiersze, a drugi to kolumny
```

Można również pisać „zapytania”. Znak & odpowiada za AND, a | odpowiada za OR:

```
df[(df.Amount > 1)  |  (df.Id < 3)]
```

Lab

1. Zainstaluj PANDAS.
2. Następnie w Pythonie zaimportuj moduł PANDAS nadając mu alias pd.
3. Poniższą komendą wczytaj dane dotyczące rankingu uczelni na świecie (plik znajdziesz w materiałach kursowych lub na https://www.kaggle.com/datasets/mylesoneill/world-university-rankings - ale stronki lubią się zmieniać, więc rekomenduję użycie dołączonych plików):

```
uni = pd.read_csv('./data/WorldUniversityRanking.csv', encoding='UTF-8')
```

4. Wyświetl nagłówek DataFrame uni korzystając z polecenia:

```
uni.head()
```

5. Wyświetl ranking uniwersytetów z Polski (kolumna **country** ma mieć wartość Poland).
6. Do kryteriów dodaj warunek powodujący wyświetlenie tylko danych za rok 2015 (kolumna **year**)
7. Do kryteriów dodaj warunek powodujący wyświetlenie tylko tych uczelni, które mają więcej niż 700 publikacji (kolumna **publications**)
8. Zmodyfikuj ostatnio dodaną część kryteriów. Uczelnia ma być wyświetlona, jeśli liczba publikacji jest większa od 700 lub liczba patentów jest większa od 300 (kolumna **patents**).

Propozycja rozwiązania – pobierz kod

```
# installation of PANDAS:
#   pip install pandas


import pandas as pd


uni = pd.read_csv('./data/WorldUniversityRanking.csv', encoding='UTF-8')


uni.head()


uni[uni.country == 'Poland']


uni[(uni.country == 'Poland') & (uni.year == 2015)]


uni[(uni.country == 'Poland') & (uni.year == 2015) & (uni.publications > 700)]


uni[(uni.country == 'Poland') & (uni.year == 2015) &
             ((uni.publications > 700) | (uni.patents > 300))]
```

Numpy i macierze

Notatka

Numpy to moduł dostarczający zoptymalizowane funkcje pozwalające na wykonywanie obliczeń matematycznych, w szczególności operacji na macierzach. Instalację modułu wykonuje się przez

```
pip install numpy
```

Nowe obiekty można tworzyć budując najpierw listę list, a następnie przekazywać ją do np.array:

```
lili = [
    [1, 2, 3], [4, 5, 6], [7, 8, 9]
]
m3x3 = np.array(lili)
```

Na obiektach ndarray, można wykonywać operacje, np. transpozycji, wykonywania typowych operacji matematycznych, liczenie niektórych funkcji np.sin:

```
m3x3.transpose()

m3x1 = np.array([10,20,0])

m3x3 * m3x1
pm3x3
m3x3[1:,:2]
```

Lab

1. Zainstaluj moduł numpy (być może już jest zainstalowany po poprzednim ćwiczeniu)
2. Załaduj moduł numpy
3. Utwórz obiekt m_1 typu np.array w oparciu o listę:

```
list_1 = [
    [10, 20, 30],
    [40, 50, 60],
    [70, 80, 90]
]
```

4. W zmiennej m_2 zapisz wynik dzielenia m_1 przez 0.333
5. W zmiennej m_sum, zapisz sumę m_1 i m_2
6. Korzystając z funkcji np.array.round, wyświetl wartości z m_sum zaokrąglone do całości.

Propozycja rozwiązania – pobierz kod

```
# installation of Numpy:
#   pip install numpy


import numpy as np


list_1 = [
    [10, 20, 30],
    [40, 50, 60],
    [70, 80, 90]
]


m_1 = np.array(list_1)
print(m_1)


m_2 = m_1 / 0.333
print(m_2)



m_sum = m_1 + m_2



print(np.round(m_sum))
```

Sortowanie bąbelkowe (buble sort)

Notatka

Sortowanie bąbelkowe polega na porównywaniu par wartości z listy. Jeśli pierwsza wartość jest większa niż kolejna, to zamieniamy je miejscami. Takie porównanie należy wykonać dla wszystkich par wartości w liście. Po wykonaniu porównań dla wszystkich par mamy pewność, że na końcu listy znajduje się największa wartość z listy. Aby posortować całą listę, w najgorszym przypadku należy wykonać tą czynność tyle razy, ile jest elementów na liście. Powoduje to wysoką złożoność obliczeniową sortowania. Ponieważ trzeba n razy wykonać n porównań, to złożoność wyraża się wzorem $O(n) = n^2$.

Algorytmy można optymalizować zauważając pewne logiczne skutki wcześniej wykonywanych operacji:

- Sortowanie można przerwać, jeśli przy poprzednim przebiegu po sortowanych danych nie wykonano żadnej zmiany
- W sortowaniu bąbelkowym po zakończonym przebiegu, na końcu listy znajduje się wartość największa. Nie ma sensu uwzględniać jej przy kolejnym przebiegu.

Lab

1. Sprawdzimy, czy ma sens optymalizowanie mało wydajnych algorytmów jak to nieszczęsne sortowanie bąbelkowe. Utworzymy dużą listę wypełnioną wartościami losowymi. Posortujemy ją korzystając z funkcji sortującej sort_bubble i sort_bubble2 zaprezentowanych na lekcji i porównamy, w jakim czasie te funkcje wykonają swoje zadanie.
2. Zaimportuj moduł time, random, zadeklaruj maksymalną długość sortowanej listy max_limit. (Na moim komputerze max_limit ustawiłem na 10000 uzyskując czas 17 i 19 sekund – dopasuj tą wartość do mocy swojej maszyny). Wygeneruj listę wartości losowych o długości max_limit. Utwórz drugą kopię tych danych:

```
import time
import random


#declarations
max_limit = 10000

my_list1 = [random.randint(0, max_limit) for i in range(max_limit)]
my_list2 = my_list1.copy()
```

3. Zadeklaruj funkcje sortujące sort_bubble i sort_bubble2 (jeśli nie chce Ci się ich pisać, to znajdziesz je w rozwiązaniu)
4. W zmiennej start zapisz aktualny czas, uruchom sortowanie przez sort_bubble, następnie w zmiennej stop zapis ponownie aktualny czas. Wyświetl różnicę między stop i start.
5. To samo wykonaj dla funkcji sort_bubble2

```
start = time.time()
sort_bubble(my_list1)
stop = time.time()
print(f'Sorting duration for function sort_bubble:    {stop - start}')
```

6. No i jak? Warto optymalizować? Zachowaj rozwiązanie do następnego LAB-a.

Propozycja rozwiązania – pobierz kod

```
import time
import random


#declarations
max_limit = 10000

my_list1 = [random.randint(0, max_limit) for i in range(max_limit)]
my_list2 = my_list1.copy()

# functions

def sort_bubble(list):

    is_change = True

    while is_change:
        is_change = False
        for i in range(len(list)-1):
            if list[i] > list[i+1]:
                list[i],list[i+1] = list[i+1],list[i]
                is_change = True



def sort_bubble2(list):

    # don't compare values at the end of the list - they are already sorted
    max_index = len(list) - 1

    for max_not_sorted_index in range(max_index,0,-1):
        is_change = False
        for i in range(max_not_sorted_index):
            if list[i] > list[i+1]:
                list[i],list[i+1] = list[i+1],list[i]
                is_change = True

        if not is_change:
            break



# checking time - case sort_bubble

start = time.time()
sort_bubble(my_list1)
stop = time.time()
print(f'Sorting duration for function sort_bubble:    {stop - start}')


# checking time - case sort_bubble2

start = time.time()
sort_bubble2(my_list2)
stop = time.time()
print(f'Sorting duration for function sort_bubble2:   {stop - start}')
```

Sortowanie przez wstawianie (insert sorting)

Notatka

Sortowanie przez wstawianie, ma na celu organicznie liczby podstawień, bo liczba porównań jest mniej więcej taka sama, jak przy sortowaniu bąbelkowym.

Sortowanie polega na wyodrębnieniu po lewej stronie części listy, którą uważamy za posortowaną. Na początku ta wyodrębniona i posortowana lista ma tylko jeden element. Następnie, dodając do tej listy nowy element, sprawdza się, gdzie powinien on być wstawiony. Wartości, które już znajdują się w uporządkowanej części listy są przesuwane w prawo, dzięki czemu robi się miejsce dla nowo dodawanego elementu.

Lab

1. Porównajmy działanie algorytmu bąbelkowego i przez wstawianie
2. Do rozwiązania poprzedniego LAB-a dodaj trzecią listę – kopię pierwszej listy.

```
...

my_list1 = [random.randint(0, max_limit) for i in range(max_limit)]
my_list2 = my_list1.copy()
my_list3 = my_list1.copy()
...
```

3. Zadeklaruj funkcję sortującą sort_insert (jeśli nie chce Ci się jej pisać, to znajdziesz ją w rozwiązaniu)
4. Zmierz czas wykonania tej funkcji

```
start = time.time()
sort_insert(my_list3)
stop = time.time()
print(f'Sorting duration for function sort_insert:    {stop - start}')
```

5. No i jak? Warto optymalizować? Oto wyniki z mojego komputera:

```
Sorting duration for function sort_bubble:    43.394434213638306
Sorting duration for function sort_bubble2:   31.159193515777588
Sorting duration for function sort_insert:    15.069791793823242
```

6. Zachowaj rozwiązanie do następnego LAB.

Propozycja rozwiązania – pobierz kod

```
import time
import random


#declarations
max_limit = 10000

my_list1 = [random.randint(0, max_limit) for i in range(max_limit)]
my_list2 = my_list1.copy()
my_list3 = my_list1.copy()

# functions


# . . .
# SKIPPING FUNCTIONS FOR BUBLE SORTING - TAKE CODE FROM PREVIOUS LESSON
# . . .


def sort_insert(list):

    # starting from 1, because this first element is already "sorted"
    # all items need to be put in place, so this loop needs to iterate for every
element
    for sort_border in range(1,len(list)):
        # we will try to put the new item into the sublist of sorted elements
        curr_idx = sort_border - 1
        value = list[curr_idx+1] # next value to be inserted into data

        while list[curr_idx] > value and curr_idx >= 0:
            list[curr_idx+1] = list[curr_idx]
            curr_idx = curr_idx-1

        #now the good position has been found and we put there the considered element
        list[curr_idx+1] = value


# checking time - case sort_bubble

start = time.time()
sort_bubble(my_list1)
stop = time.time()
print(f'Sorting duration for function sort_bubble:    {stop - start}')


# checking time - case sort_bubble2

start = time.time()
sort_bubble2(my_list2)
stop = time.time()
print(f'Sorting duration for function sort_bubble2:   {stop - start}')


# checking time - case  sort_insert

start = time.time()
sort_insert(my_list3)
stop = time.time()
print(f'Sorting duration for function sort_insert:    {stop - start}')
```

Sortowanie przez wybieranie (select sorting)

Notatka

W sortowaniu przez wybieranie, należy wykonać tyle „przebiegów”, co lista ma elementów. Celem każdej iteracji jest znalezienie kolejnej wartości, która jeszcze nie została przepisana do posortowanej listy.

W zerowym przebiegu (trzymając się numeracji pythonowej od zera) należy znaleźć najmniejszą wartość z całej listy i zapisać ją na zerowej pozycji. W pierwszym przebiegu szukamy najmniejszej wartości na liście, ale pomijamy element zerowy (bo ten element jest już na właściwym miejscu). Po znalezieniu tej wartości umieszczamy ją na pozycji 1. Następnie szukamy wartości do umieszczenia na pozycji nr 2. W tym celu przeglądamy wszystkie jeszcze nieuporządkowane wartości szukając wśród nich najmniejszej. Ten cykl powtarza się n-razy dla listy zawierającej n elementów.

Lab

1. Porównajmy działanie omówionych do tej pory algorytmów.
2. Do rozwiązania poprzedniego LAB-a dodaj czwartą listę – kopię pierwszej listy.

```
...

my_list1 = [random.randint(0, max_limit) for i in range(max_limit)]
my_list2 = my_list1.copy()
my_list3 = my_list1.copy()
my_list4 = my_list1.copy()
...
```

3. Zadeklaruj funkcję sortującą sort_select (jeśli nie chce Ci się jej pisać, to znajdziesz ją w rozwiązaniu)
4. Zmierz czas wykonania tej funkcji

```
start = time.time()
sort_select(my_list4)
stop = time.time()
print(f'Sorting duration for function sort_select:    {stop - start}')
```

5. No i jak? Warto optymalizować? Oto wyniki z mojego komputera dla listy 10000 wartości – dopasuj długość listy, tak żeby Ci się nie nudziło 😊:

```
Sorting duration for function sort_bubble:    28.233525037765503
Sorting duration for function sort_bubble2:   18.992961645126343
Sorting duration for function sort_insert:    10.460476160049438
Sorting duration for function sort_select:    8.868878841400146
```

Propozycja rozwiązania – pobierz kod

```
import time
import random


#declarations
max_limit = 10000

my_list1 = [random.randint(0, max_limit) for i in range(max_limit)]
my_list2 = my_list1.copy()
my_list3 = my_list1.copy()
my_list4 = my_list1.copy()

# functions


# . . .
# SKIPPING FUNCTIONS FOR OTHER ALGORITHMS – TAKE CODE FROM PREVIOUS LESSON
# . . .



def sort_select(list):

    for run in range(len(list)):
        min_index = run
        for i in range(run+1, len(list)):
            if list[i] < list[min_index]:
                min_index = i
        list[run], list[min_index] = list[min_index], list[run]


# checking time - case sort_bubble

start = time.time()
sort_bubble(my_list1)
stop = time.time()
print(f'Sorting duration for function sort_bubble:    {stop - start}')


# checking time - case sort_bubble2

start = time.time()
sort_bubble2(my_list2)
stop = time.time()
print(f'Sorting duration for function sort_bubble2:   {stop - start}')


# checking time - case  sort_insert

start = time.time()
sort_insert(my_list3)
stop = time.time()
print(f'Sorting duration for function sort_insert:    {stop - start}')


# checking time - case  sort_select

start = time.time()
sort_select(my_list4)
stop = time.time()
print(f'Sorting duration for function sort_select:    {stop - start}')
```

Sortowanie przez scalanie (merge sort)

Notatka

Sortowanie przez scalanie bazuje na idei „dziel i rządź”. Najpierw zbiór danych dzieli się na połowy, tak długo, aż wszystkie te zbiory będą jednoelementowe. Takie trywialne listy można oczywiście traktować jako posortowane.

W drugim kroku łączy się te zbiory w parach, w taki sposób, że pobranie kolejnej wartości odbywa się z listy, w której występuje mniejsza wartość. W ten sposób z list jednoelementowych powstają listy dwuelementowe, z list dwuelementowych robią się listy czteroelementowe itd. Każda z takich powstających list jest już jednak posortowana.

Lab

1. Porównajmy działanie algorytmu sortowania przez scalanie z poprzednio poznanymi algorytmami
2. Do rozwiązania poprzedniego LAB-a dodaj piątą listę – kopię pierwszej listy.

```
...
my_list5 = my_list1.copy()
...
```

3. Zadeklaruj funkcję sortującą sort_merge (jeśli nie chce Ci się jej pisać, to znajdziesz ją w rozwiązaniu)
4. Zmierz czas wykonania tej funkcji

```
start = time.time()
sort_merge(my_list5)
stop = time.time()
print(f'Sorting duration for function sort_insert:    {stop - start}')
```

5. No i jak? Warto optymalizować? W celu lepszej obserwacji postępów, możesz rozważyć zwiększenie liczby sortowanych wartości do np. 20000, ale wszystko zależy od mocy Twojego komputera i czasu, jaki podczas działania programu chcesz przepalić na Facebooku i temu podobnych 😊. Oto wyniki z mojego komputera:

```
Sorting duration for function sort_bubble:    115.60329413414001
Sorting duration for function sort_bubble2:   75.78087019920349
Sorting duration for function sort_insert:    40.83059573173523
Sorting duration for function sort_select:    39.505269289016724
Sorting duration for function sort_merge:     0.447995662689209
```

6. Zachowaj rozwiązanie do kolejnego LAB

Propozycja rozwiązania – pobierz kod

```
    import time
import random


#declarations
max_limit = 10000

my_list1 = [random.randint(0, max_limit) for i in range(max_limit)]
my_list2 = my_list1.copy()
my_list3 = my_list1.copy()
my_list4 = my_list1.copy()
my_list5 = my_list1.copy()


# functions


# . . .
# SKIPPING OTHER FUNCTIONS - TAKE CODE FROM PREVIOUS LESSON
# . . .


def sort_merge(list):

    list_len = len(list)
    sorted_list = []

    if list_len <= 1:
        sorted_list = list
    else:
        middle_point = list_len // 2
        list_left = sort_merge(list[:middle_point])
        list_right = sort_merge(list[middle_point:])

        idx_left = idx_right = 0
        while  idx_left < len(list_left) and idx_right < len(list_right):
            if list_left[idx_left] < list_right[idx_right]:
                sorted_list.append(list_left[idx_left])
                idx_left += 1
            else:
                sorted_list.append(list_right[idx_right])
                idx_right += 1

        sorted_list.extend(list_left[idx_left:])
        sorted_list.extend(list_right[idx_right:])

    return sorted_list


# checking time - case sort_bubble

# SKIPPING PART OF OTHER TESTS

# checking time - case  sort_insert

start = time.time()
sort_merge(my_list5)
stop = time.time()
print(f'Sorting duration for function sort_merge:    {stop - start}')
```

Szybkie sortowanie (quick sort)

Notatka

Quick sort stosuje podobną sztuczkę, co algorytm merge sort. Dane do posortowania są najpierw dzielone na coraz to mniejsze podzbiory, które będą układane w kolejnych rekurencyjnych wywołaniach funkcji sortującej.

W tym algorytmie każdorazowo wybiera się jeden element listy, względem, którego jest wykonywany podział. W najprostszym przypadku może to być np. ostatni element tej listy. Następnie lista jest przeglądana wartość po wartości i jeśli analizowany element jest mniejszy od wybranej wartości podziału, to należy go przesunąć na lewo.

W pracy algorytmu stosuje się pomocnicze wskaźniki. Jeden z nich wskazuje na w danej chwili analizowany element (j), a drugi (i) wskazuje na punkt podziału wartości mniejszych i większych niż wybrany wcześniej punkt podziału.

Lab

1. Porównajmy działanie algorytmu sortowania quick sort z poprzednio poznanymi algorytmami
2. Do rozwiązania poprzedniego LAB-a dodaj szóstą listę – kopię pierwszej listy.

```
...

my_list6 = my_list1.copy()
...
```

3. Zadeklaruj funkcję sortującą quick_sort (jeśli nie chce Ci się jej pisać, to znajdziesz ją w rozwiązaniu)
4. Zmierz czas wykonania tej funkcji

```
start = time.time()
sort_quick(my_list6, 0, len(my_list6)-1)
stop = time.time()
print(f'Sorting duration for function sort_insert:    {stop - start}')
```

5. No i jak? Warto optymalizować? Oto wyniki z mojego komputera:

```
Sorting duration for function sort_bubble:    121.02622604370117
Sorting duration for function sort_bubble2:   78.53391003608704
Sorting duration for function sort_insert:    39.83429741859436
Sorting duration for function sort_select:    34.25660419464111
Sorting duration for function sort_merge:     0.2353534698486328
Sorting duration for function quick_sort:     0.14484691619873047
```

Propozycja rozwiązania – pobierz kod

```
import time
import random

#declarations
max_limit = 20000

my_list1 = [random.randint(0, max_limit) for i in range(max_limit)]
my_list2 = my_list1.copy()
my_list3 = my_list1.copy()
my_list4 = my_list1.copy()
my_list5 = my_list1.copy()
my_list6 = my_list1.copy()


# functions

# . . .
# SKIPPING OTHER FUNCTIONS - TAKE CODE FROM PREVIOUS LESSON
# . . .


def quick_sort_divide(list, start, end):

    i = start
    border_value = list[end]

    for j in range(start, end):

        if list[j] <= border_value:
            list[i], list[j] = list[j], list[i]
            i += 1

    list[i], list[end] = list[end], list[i]
    return i


def quick_sort(list, start, end):

    if start < end:

        border_index = quick_sort_divide(list, start, end)
        quick_sort(list, start, border_index -1)
        quick_sort(list, border_index + 1, end)


# . . .
# SKIPPING OTHER CHECKS - TAKE CODE FROM PREVIOUS LESSON
# . . .


# checking time - case  quick_sort

start = time.time()
quick_sort(my_list6, 0, len(my_list6)-1)
stop = time.time()
print(f'Sorting duration for function quick_sort:     {stop - start}')
```

Wyszukiwanie liniowe

Notatka

Wyszukiwanie liniowe, to dość trywialny sposób przeglądania listy w celu odnalezienia interesującej nas wartości. Trzeba po prostu przejść przez wszystkie elementy listy i porównywać je z szukaną wartością. Jeśli dany element jest równy szukanej wartości, to należy go zwrócić.

Warto zauważyć, że:

- Lista nie musi być uporządkowana
- Jeśli mamy pewność, że elementy listy są unikalne, to można zakończyć przeszukiwanie po znalezieniu pierwszej wartości
- Jeśli element nie występuje na liście, to trzeba przejrzeć całą listę
- Algorytm ma złożoność obliczeniową O(n) – masz n elementów na liście, to musisz wykonać n porównań

Lab

1. Napisz funkcję wyszukiwania liniowego, która odnajdzie i wyświetli wszystkie informacje o osobie z określonym numerem telefonu. Oto lista z danymi do przeszukania:

```
tel_list = [
    {'number': 123123123, 'name': 'photovoltaic'},
    {'number': 789789789, 'name': 'subscription of magazines'},
    {'number': 567567567, 'name': 'lottery'},
    {'number': 345345345, 'name': 'show of pots'},
    {'number': 678678678, 'name': 'grandson asking for money'},
    {'number': 234234234, 'name': 'loans'},
    {'number': 456456456, 'name': 'free medical examinations'}
]
```

Propozycja rozwiązania – pobierz kod

```
tel_list = [
    {'number': 123123123, 'name': 'photovoltaic'},
    {'number': 789789789, 'name': 'subscription of magazines'},
    {'number': 567567567, 'name': 'lottery'},
    {'number': 345345345, 'name': 'show of pots'},
    {'number': 678678678, 'name': 'grandson asking for money'},
    {'number': 234234234, 'name': 'loans'},
    {'number': 456456456, 'name': 'free medical examinations'}
]

def search_linear(number, list_of_dicts):

    idx = 0
    found = False

    while not found and idx < len(list_of_dicts):
        if list_of_dicts[idx]['number'] == number:
            found = True
        else:
            idx += 1

    return idx if idx < len(list_of_dicts) else None


number = 567567567
idx = search_linear(number, tel_list)
print(f'The phone number {number} details are under index: {idx}')

if idx:
    print(tel_list[idx])
```

Wyszukiwanie binarne

Notatka

Wyszukiwanie binarne jest wykonywane na danych posortowanych. Polega na tym, że w każdej iteracji wybierany jest punkt środkowy między wartościami start i stop wskazującymi początkowo na pierwszy i ostatni element listy. Następnie sprawdza się, czy wybrany punkt środkowy ma wartość równą poszukiwanej wartości. Jeśli tak, to można zakończyć wyszukiwanie. Jeśli nie, to:

- Gdy poszukiwana wartość jest mniejsza niż wartość w punkcie środkowym, to należy przesunąć end w lewo, tuż przed wartością środkową;
- Gdy poszukiwana wartość jest większa niż wartość w punkcie środkowym, to należy przesunąć start w prawo, tuż za wartością środkową;

W ten sposób w każdej iteracji zakres poszukiwanych wartości jest ograniczany o połowę, co znacznie przyśpiesza wyszukiwanie. Kończy się ono sukcesem, gdy wartość wskazywana przez punkt środkowy jest równa poszukiwanej wartości. Wyszukiwanie kończy się porażką, gdy start jest mniejszy niż end.

Lab

1. Porównamy sprawność działania wyszukiwania liniowego i binarnego.
2. Przygotuj funkcję search_linear(value, list) oraz search_binary(value, list) implementujące wyszukiwanie liniowe i binarne
3. Utwórz listę ordered_list, np z 100.000.000 kolejnych liczb (liczbę dopasuj do mocy swojego komputera)
4. Wylosuj wartość z zakresu 0 – 100.000.000, wyświetl ją – będzie to szukana wartość.
5. Uruchom funkcje mierząc ich czas wykonania.
6. Moje wyniki przy jednej z prób to:

```
Searching value 87355651
Linear search: value found at 87355651 in 28.921273469924927 seconds
Binary search: value found at 87355651 in 0.0 seconds
```

Wartość “0” oznacza, że zmierzony czas był na tyle mały, że funkcja time.time() sobie z nim nie poradziła 😄. W takim przypadku można użyć time.perf_counter(), który jest dokładniejszy:

```
Searching value 96136021
Linear search: value found at 96136021 in 30.857310000000002 seconds
Binary search: value found at 96136021 in 9.030000000365135e-05 seconds
```

Propozycja rozwiązania – pobierz kod

```
def search_linear(value, list):

    idx = 0
    found = False

    while not found and idx < len(list):
        if list[idx] == value:
            found = True
        else:
            idx += 1

    return idx if idx < len(list) else None


def search_binary(value, list):

    start = 0
    end = len(list)-1
    found = False

    while start <= end and not found:
        mid = (start + end) // 2
        if list[mid] == value:
            found = True
        else:
            if value < list[mid]:
                end = mid - 1
            else:
                start = mid + 1

    if found:
        return mid
    else:
        return None


import time
import random

max_len = 100000000
ordered_list = list(range(max_len))
value = random.randint(0, max_len)

print(f'Searching value {value}')

# checking time - linear search

start = time.perf_counter()
idx = search_linear(value, ordered_list)
stop = time.perf_counter()
print(f'Linear search: value found at {idx} in {stop - start} seconds')


# checking time - binary search

start = time.perf_counter()
idx = search_binary(value, ordered_list)
stop = time.perf_counter()
print(f'Binary search: value found at {idx} in {stop - start} seconds')
```

Wyszukiwanie przez interpolację

Notatka

Interpolation Search wykorzystuje mniej więcej ten sam mechanizm szukania, co binary search. Jedną różnicą jest wyznaczanie wartości środkowej. Tutaj robimy to następującą formułą:

$$mid = start + \frac{end - start}{list[end] - list[start]} * (\text{searchValue} - \text{list[start]})$$

Algorytm dobrze sprawdza się, gdy dane zapisane w liście są rozłożone liniowo, w przeciwnym razie, wydajność może spaść.

Lab

1. Porównajmy wydajność algorytmu wyszukiwania binarnego i przez interpolację.
2. Przygotuj funkcje search_binary(value, list) oraz search _interpolate(value, list).
3. Tak, jak w poprzednim LAB, utwórz listę ordered_list, np z 100.000.000 kolejnych liczb (liczbę dopasuj do mocy swojego komputera)
4. Wylosuj wartość z zakresu 0 – 100.000.000, wyświetl ją – będzie to szukana wartość.
5. Uruchom funkcje mierząc ich czas wykonania. Do mierzenia czasu wykorzystaj funkcję time.perf_counter_ns() – zwraca ona wynik w nanosekundach, co może tutaj bardzo pomóc.
6. Moje wyniki przy jednej z prób to:

```
Searching value 68230637
Binary search: value found at 68230637 in 78200 nano seconds
Interpolation search: value found at 68230637 in 44400 nano seconds
```

7. Pamiętaj, że wynik funkcji interpolującej będzie tym lepszy im bardziej liniowo są rozłożone wartości w analizowanej liście.

Propozycja rozwiązania – pobierz kod

```
def search_binary(value, list):

    start = 0
    end = len(list)-1
    found = False

    while start <= end and not found:
        mid = (start + end) // 2
        if list[mid] == value:
            found = True
        else:
            if value < list[mid]:
                end = mid - 1
            else:
                start = mid + 1

    if found:
        return mid
    else:
        return None


def search_interpolate(value, list):

    start = 0
    end = len(list) - 1
    found = False

    while start <= end and value >= list[start] and value <= list[end]:
        # Find the mid point
        if list[start] != list[end]:
            mid = start + int(  ( float(end - start) / (list[end] - list[start]) )
                              * (value - list[start]) )
        else:
            mid = start

        # Compare the value at mid point with search value
        if list[mid] == value:
            found = True
            break
        else:
            if value < list[mid]:
                end = mid - 1
            else:
                start = mid + 1

    if found:
        return mid
    else:
        return None

import time
import random

max_len = 100000000
ordered_list = list(range(max_len))
value = random.randint(0, max_len)
print(f'Searching value {value}')

# checking time - binary search
start = time.perf_counter_ns()
idx = search_binary(value, ordered_list)
stop = time.perf_counter_ns()
print(f'Binary search: value found at {idx} in {stop - start} nano seconds')


# checking time - interpolate search
start = time.perf_counter_ns()
idx = search_interpolate(value, ordered_list)
stop = time.perf_counter_ns()
print(f'Interpolation search: value found at {idx} in {stop - start} nano seconds')
```

Quick Select

Notatka

Algorytm Quick Select pozwala na wybranie k-tego elementu zbioru nieposortowanego. Dzięki zastosowaniu algorytmu Quick Sort względem tylko tej części danych, w których znajduje się k-ty element, do wykonania zadania nie jest potrzebne sortowanie całego zbioru danych, a tylko jego części.

Korzystamy tu z tego, że podczas sortowania przez Quick Sort, w każdym kroku jest ustalana docelowa pozycja jednego z elementów listy. Wartości mniejsze niż ten element są umieszczane po lewej stronie, a wartości większe – po prawej. Algorytm sortujący musiałby w następnym kroku posortować zarówno lewą jak i prawą stronę, ale ponieważ Quick Select skupia się na wybraniu tylko jednego elementu, to wystarczy uruchomienie sortowania tylko tej części, w której znajduje się poszukiwana wartość.

Lab

1. Zgodnie z intuicyjnym opisem na Wikipedii https://pl.wikipedia.org/wiki/Mediana mediana to:
   W danym szeregu uporządkowanym liczba, która jest w połowie szeregu w wypadku nieparzystej liczby elementów. Dla parzystej liczby elementów – średnia arytmetyczna dwóch środkowych liczb.
2. Korzystając z Quick Select wyznacz medianę z dowolnej listy z wartościami numerycznymi.
3. Przetestuj działanie funkcji na następujących listach:

```
list_even = [1000,2,30,6,4,70,800,90,1,50]
list_odd = [1000,2,30,6,4,70,800,90,1]
```

Propozycja rozwiązania – pobierz kod

```
def divide(list, start, end):

    i = start
    border_value = list[end]

    for j in range(start, end):

        if list[j] <= border_value:
            list[i], list[j] = list[j], list[i]
            i += 1

    list[i], list[end] = list[end], list[i]
    return i


def quick_select(list, start, end, k_th):

    print(f'DEBUG: {list}')

    if k_th >= len(list):
        return None

    if start <= end:

        border_index = divide(list, start, end)

        if border_index == k_th:
            print(f'DEBUG: {list} - found: {list[border_index]}')
            return list[border_index]
        elif border_index > k_th:
            return quick_select(list, start, border_index -1, k_th)
        else:
            return quick_select(list, border_index + 1, end, k_th)


def median(list):

    if len(list) %2 == 1:
        return quick_select(list, 0, len(list)-1, len(list)//2 )
    else:
        m1 = quick_select(list, 0, len(list)-1, len(list)//2-1 )
        m2 = quick_select(list, 0, len(list)-1, len(list)//2 )
        return (m1 + m2) / 2



list_even = [1000,2,30,6,4,70,800,90,1,50]
print(median(list_even))

list_odd = [1000,2,30,6,4,70,800,90,1]
print(median(list_odd))
```

Wyszukiwanie wzorca w tekście – Knuth-Morris-Pratt

Notatka

Algorytm KMP wyszukiwania wzorca w tekście optymalizuje odnajdowanie pozycji wystąpienia wzorca w oryginalnym tekście poprzez wstępną analizę wzorca. Algorytm buduje najpierw funkcję/tablicę przesunięcia, w której sprawdza, czy początkowe litery wzorca pojawiają się jeszcze gdzieś dalej w tym wzorcu. Jeśli tak, to każdej literce powtórzonego prefiksu wzorca przypisuje liczbę powtórzonych znaków, np.

S	Q	S	Q	R
0	0	1	2	0

Podczas porównywania znaków tekstu i wzorca:

- Jeśli znaki są zgodne, to indeks wskazujący analizowany tekst i wzorzec przesuwają się o 1 dalej
- Jeśli znaki są różne, to indeks wskazujący aktualną literkę wzorca przesuwa się do znaku odczytanego dla ostatnio dobrze porównanej litery z funkcji przesunięcia, o ile tylko ten indeks już nie wskazuje na 0
- Jeśli znaki są różne, ale indeks wskazujący aktualną literkę wzorca jest już na początku wzorca, to wtedy należy przejść do sprawdzania kolejnej litery tekstu.

Lab

1. Porównajmy szybkość działania algorytmu KMP z algorytmem naiwnym. Uwaga – uzyskiwane wyniki będą zmieniać się przy każdym uruchomieniu programu. Sprawdź działanie programu kilka razy. W niektórych przypadkach może się okazać, że algorytm naiwny działa lepiej
2. Napisz samodzielnie funkcję wyszukującą wzorzec w tekście metodą naiwną (no dobrze – jak bardzo nie chcesz, to znajdziesz ją w rozwiązaniu poniżej). Niech ta funkcja zwraca listę odnalezionych indeksów, gdzie oba napisy się do siebie dopasowały.
3. Napisz funkcję dopasowującą wzorzec z wykorzystaniem algorytmu KMP. (Też jest w rozwiązaniu poniżej). Niech ta funkcja zwraca listę znalezionych indeksów, tak samo, jak robiła to funkcja wyszukiwania naiwnego.
4. Utwórz losowy text o długości 1.000.000 powtarzający litery a i b i wyszukaj w nim wzorca abababababaa.
5. Porównaj czas działania obu funkcji
6. W moim przypadku najdłużej trwało… generowanie ciągu znaków, ale jak już został on wygenerowany to uzyskałem wynik:

```
Generating random text
Text generation:    7.549620866775513
Searching
Naive search: found 76 matches in 0.7837777137756348 seconds
KMP search:   found 76 matches in 0.6624813079833984 seconds
```

Propozycja rozwiązania -pobierz kod

```
def generate_prefix(pattern):
    pat_len = len(pattern)
    pat_fun = [0] * pat_len
    pref_len = 0    # length of the previous longest prefix suffix

    if pat_len <= 1:   # eliminate trivial conditions
        return pat_fun

    pat_fun[0] = 0     # always 0
    i = 1

    while i < pat_len:
        if pattern[i] == pattern[pref_len]:
            pref_len += 1
            pat_fun[i] = pref_len
            i += 1
        else:
        # This is tricky. The pattern may contain multiple sub-prefixes. This allows to use them
            if pref_len != 0:
                pref_len = pat_fun[pref_len-1]
                # we do not increment i here
                # the loop will execute again for the same i and pref_len
                # until a matching prefix would be found or we step down to pref_len == 0
            else:
                pat_fun[i] = 0
                i += 1
    return pat_fun


def search_kmp(pattern, text):
    pat_len = len(pattern)
    txt_len = len(text)
    pat_fun = generate_prefix(pattern)

    results_list = []

    p = 0 # index pointing to pattern
    t = 0 # index pointing to text

    while t < txt_len:
        if pattern[p] == text[t]:
            t += 1
            p += 1
            if p == pat_len:
                results_list.append(t-p)
                p = pat_fun[p-1]
        else:
            if p != 0:
                p = pat_fun[p-1]
            else:
                t += 1

    return results_list


def search_naive(pattern, text):

    result_list = []

    for text_start in range(len(text) - len(pattern) + 1):
        found = True
        for i in range(len(pattern)):
            if pattern[i] != text[text_start+i]:
                found = False
                break
        if found:
            result_list.append(text_start)

    return result_list
```

```
import random
import time

max_len = 1000000
allowed_chars = ['a', 'b']

print('Generating random text')
start = time.time()
text = ''.join(random.choice(allowed_chars) for i in range(max_len))
stop = time.time()
print(f'Text generation:    {stop - start}')

pattern = 'abababababab'

print('Searching')
# checking time - naive search

start = time.time()
results_naive = search_naive(pattern, text)
stop = time.time()
print(f'Naive search: found {len(results_naive)} matches in {stop - start} seconds')


# checking time - KMP search

start = time.time()
results_kmp = search_kmp(pattern, text)
stop = time.time()
print(f'KMP search:   found {len(results_kmp)} matches in {stop - start} seconds')
```

Ta strona celowo jest pusta

Wyszukiwanie wzorca w tekście – Rabin-Karp

Notatka

Hash zwany też skrótem pozwala na przypisanie pewnym wartościom (liczbom, napisom, a nawet plikom) jednej, zazwyczaj mniejszej (krótszej) wartości. Przykładem prostej funkcji hashującej jest przypisanie liczbie hasha wyznaczonego jako reszta z dzielenia przez 10. Równość wartości hash nie oznacza, że oryginalne wartości są sobie równe jednak, jeśli oryginalne wartości są równe, to mają taki sam hash.

Algorytm Rabin-Karpa wyznacza wartość hash dla poszukiwanego wzorca. Następnie wyznacza wartość hash, dla każdego ciągu znaków o długości równej długości wzorca, występującego w tekście. Dzięki temu, zamiast porównywać występowanie liter we wzorcu i w tekście wystarcza sprawdzać, czy istnieją dla tekstu takie wartości hash, które są równe hashowi wzorca.

Gdy taki pasujący hash zostanie odnaleziony, to nie oznacza to, że wystąpiło dopasowanie, bo kilka różnych tekstów może zwrócić tą samą wartość hash. Dlatego w ostatnim kroku należy literka po literce sprawdzić, czy wzorzec w całości pasuje do liter tekstu w „podejrzanym" fragmencie tekstu.

Lab

1. Porównajmy sprawność algorytmów z po przedniego LAB, z przedstawionym w tej lekcji algorytmem Rabin-Karp
2. Dodaj do skryptu z poprzedniego zadania definicję funkcji generującej hash oraz implementującą wyszukiwanie algorytmem Rabin-Karp
3. Dodaj sekcję pomiaru czasu dla tego algorytmu.
4. Wyniki jaki uzyskałem na moim komputerze były następujące:

```
Generating random text
Text generation:    8.05894422531128
Searching
Naive search:       found 67 matches in 0.8225488662719727 seconds
KMP search:         found 67 matches in 0.6461479663848877 seconds
Rabin-Karp search: found 67 matches in 0.9379324913024902 seconds
```

5. Martwi Cię, że Rabin-Karp wypdał najsłabiej? Mnie też 😅, ale wytłumaczenie tej sytuacji znajdziesz w kolejnym lab.
6. Zapoznaj się z artykułem https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm Jest tam sekcja dyskutująca funkcję wyliczającą hash. Przeczytaj ją (w miarę możliwości ze zrozumieniem 😊)

Propozycja rozwiązania – pobierz kod

```
# definition of functions form previous lab (skipped here) - copy from previous lab



def generate_hash(text, pattern):

    len_text = len(text)
    len_pattern = len(pattern)
    # lists with ordinals for charactes from the text and pattern
    ascii_text = [ord(i) for i in text]
    ascii_pattern = [ord(i) for i in pattern]

    # calculate hash for pattern
    hash_pattern = sum(ascii_pattern)

    # calculate hash for text
    len_hash_text = len_text - len_pattern + 1
    hash_text = [0] * len_hash_text

    for i in range(0, len_hash_text):
        if i == 0:
            hash_text[i] = sum(ascii_text[:len_pattern])
        else:
            hash_text[i] = (hash_text[i-1] - ascii_text[i-1] + ascii_text[i+len_pattern-1])

    return [hash_text, hash_pattern]


def search_rabin_karp(pattern, text):
    # hash_text has length of:   len_text - len_pattern + 1
    hash_text, hash_pattern = generate_hash(text, pattern)

    len_text = len(text)
    len_pattern = len(pattern)

    result_list = []

    for i in range(len(hash_text)):

        if hash_text[i] == hash_pattern:   # it is possible that we have a match
            matched_chars_count = 0
            for j in range(len_pattern):
                if pattern[j] == text[i+j]:
                    matched_chars_count += 1
                else:
                    break                  # no - it is not a match, aborting!

       # when matched_chars_count == len_pattern then the full pattern was in the text
            if matched_chars_count == len_pattern:
                result_list.append(i)

    return result_list



# measurement of time for the other functions - copy form previous lab


start = time.time()
results_rk = search_rabin_karp(pattern, text)
stop = time.time()
print(f'Rabin-Karp search: found {len(results_rk)} matches in {stop - start} seconds')
```

Wyszukiwanie wzorca w tekście – Algorytm Boyer-Moore

Notatka

Algorytm porównuje literki wzorca i tekstu jedna po drugiej, zaczynając od końca wzorca. Optymalizacja liczby porównań jest wykonywana przy pomocy dobrej i złej heurystyki:

- Dobra heurystyka – sprawdza, czy już dopasowany ciąg liter (suffix) powtarza się jeszcze gdzieś we wzorcu. Jeśli tak, to przesunięcie wzorca względem tekstu jest wykonywane tak, aby ów wcześniej dopasowany fragment tekstu pokrył się z powtórzonym suffixem wzorca
  - Jeśli takich powtórzonych suffixów we wzorcu jest więcej, to należy przesunąć się o najmniejszą liczbę pól
  - Jeśli suffix ma długość >1, i we wzorcu suffix się nie powtarzał, to należy skrócić suffix o 1 i spróbować go dopasować ponownie
- Zła heurystyka – sprawdza, czy literka tekstu, na której doszło do konfliktu, występuje we wzorcu. Jeśli nie, to można zacząć porównywanie wzorca i tekstu tuż za tą literką.

Jeśli dobra i zła heurystyka proponują przesunięcie o różną liczbę pól, to należy wybrać większą wartość.

Lab

1. Do wyników poprzedniego LAB, dodamy teraz kod implementujący algorytm Boyer-Moora i sprawdzimy, jak sobie radzi pod względem czasu wykonania.
2. Dodaj definicję algorytmu oraz sekcję pomiaru czasu, następnie uruchom program. Moje przykładowe wyniki:

```
Generating random text
Text generation:    8.18330454826355
Searching
Naive search:       found 63 matches in 1.2305684089660645 seconds
KMP search:         found 63 matches in 0.7238783836364746 seconds
Rabin-Karp search:  found 63 matches in 0.9373531341552734 seconds
Boyer-Moore search: found 63 matches in 2.1654999256134033 seconds
```

3. Dlaczego kolejne algorytmy wypadają coraz gorzej? Może dlatego, że aktualnie tekst jest generowany wyłącznie z liter ‘a’ i ‘b’, więc np. w Boyer-Moore nie mamy możliwości korzystania z bad-heuristics, a w Rabin-Karp hashe mają podobne wartości? Zmień listę allowed_chars na:

```
allowed_chars = ['a','b',' ']
```

4. Uruchom program ponownie. Moje wyniki to tym razem:

```
Generating random text
Text generation:    7.980039596557617
Searching
Naive search:       found 1 matches in 0.9169530868530273 seconds
KMP search:         found 1 matches in 0.8389942646026611 seconds
Rabin-Karp search:  found 1 matches in 0.7530367374420166 seconds
Boyer-Moore search: found 1 matches in 0.367978572845459 seconds
```

5. Jak więc widać – nie zawsze każdy algorytm spisze się na medal w każdej sytuacji. Czasami algorytm trzeba będzie właściwie dobierać do typu danych z jakimi pracujemy. To tłumaczy, dlaczego mamy tyle różnych algorytmów do rozwiązania jednego problemu i dlaczego wciąż do rozwiązywania problemów informatycznych jest potrzebny człowiek (a może z czasem AI).

Propozycja odpowiedzi – pobierz kod

```
import random
import time

max_len = 1000000


# uncomment one or the other line to make more tests:

#allowed_chars = ['a','b']
allowed_chars = ['a','b',' ']


#
# here the other algorithms should be defined - copy them form previous labs
#

def search_boyer_moore(pattern,text):

    len_text = len(text)
    len_pattern = len(pattern)
    result_list = []

    t=0
    is_match = True

    while t <= len_text - len_pattern:
        #print(f'DEBUG: --- {text}')
        #print(f'       --- {" "*t}{pattern}')
        # pattern analyzed in reverse mode
        for p in range(len_pattern-1, -1, -1):
            # mismatch! find bad and good shift
            if pattern[p] != text[t+p]:
                is_match = False
                #print(f'DEBUG: MISMATCH index {t+p}/{p}: text - {text[t+p]} pattern -
{pattern[p]}')

                # GOOD SHIFT CALCULATION
                # it is end of pattern - no good shift
                if p == len_pattern - 1:
                    good_shift = 0
                    #print(f'  DEBUG: mismatch on last character - good_shift proposed
= {good_shift}')
                else:
               # search for substrings matching to already processed suffix
               # note: it is reverse pattern processing, adding k makes suffix shorter
                    k=1
                    while text[t+p+k:t+len_pattern] not in pattern[0:len_pattern-1]:
                        k=k+1
                    # good-suffix found and is longer than 1 - use it!
                    if len(text[t+p+k:t+len_pattern]) != 1:
                        good_shift=t+p+k-pattern[0:len_pattern-
1].rfind(text[t+p+k:t+len_pattern])
                        #print(f'  DEBUG: good suffix has length > 1 - good_shift
proposed = {good_shift}')
                    else:
     # good-suffix found, but has length of 1 - don't use it, bad shift will be better
                        good_shift=0
                        #print(f'  DEBUG: good suffix has length of 1 - good_shift
proposed = {good_shift}')

                # BAD SHIFT CALCULATION
                # if bad character exists in the pattern
                if text[t+p] in pattern[0:p]:
                    # move to this character, so it matches with the pattern and text
                    bad_shift=t+p-pattern[0:p].rfind(text[t+p])
                    #print(f'  DEBUG: bad character {text[t+p]} found - bad_shift
proposed = {bad_shift}')
                else:
```

```
                    #if bad character not found - move analysis after the current char
                    bad_shift=t+p+1
                    #print(f'  DEBUG: bad character not found - bad_shift proposed =
{bad_shift}')

                t=max((bad_shift, good_shift))
                #print(f'  DEBUG: selected shift to {t}')
                break

        if is_match:
            result_list.append(t)
            #print(f'DEBUG: Match found on index {t}')
            t = t+1
        else:
            is_match = True

    return result_list


#
#  Here copy the code measuring execution time for the other algorithms
#


# checking time - Boyer-Moore search

start = time.time()
results_bm = search_boyer_moore(pattern, text)
stop = time.time()
print(f'Boyer-Moore search: found {len(results_bm)} matches in {stop - start}
seconds')
```

Ta strona celowo jest pusta

Pakowanie plecaka – metoda naiwna (brute force)

Notatka

Problem plecaka opisuje sytuację złodzieja, który ma ukraść przedmioty o największej wartości, ale musi je zmieścić w plecaku o ograniczonej wagowo pojemności. Należy zoptymalizować wartość przedmiotów mając ograniczenie związane z ich wagą.

Algorytmy naiwne rozwiązują ten problem poprzez wygenerowanie wszystkich możliwych kombinacji, odrzucenie tych, które nie spełniają warunku ograniczenia, a następnie wybór spośród poprawnych rozwiązań tego, które pozwoliło uzyskać największą wartość. Niestety, takie podejście ma wysoką złożoność obliczeniową, np. podejmując decyzję zabrać/zostawić dla n elementów, trzeba rozważyć 2^n możliwości. Złożoność obliczeniowa jest więc określona wzorem $O(n)=2^n$

Lab

1. Dana jest kwota pieniężna value oraz lista dostępnych nominałów monet:

```
value = 17
coins = [1,2,5,10,20]
```

2. Ustal, jaka jest najmniejsza liczba monet i jakie to są monety, która pozwala wydać wskazaną kwotę. Niech funkcja, którą napiszesz zwraca listę nominałów tych monet.
3. Wskazówki – na razie użyj metody naiwnej:
    a. Funkcja może przyjmować argumenty coins i value
    b. Jeśli value == 0 to należy zwrócić [] – to chyba logiczne
    c. Przygotuj pomocniczą zmienną ret_tab_coins, w której zapiszesz zwracany roboczy wynik. Początkowo zapisz w niej wartość None – co oznacza „nie ma wyniku”
    d. Dla każdego nominału monety c ze zbioru coins:
        i. Jeśli tylko value >= coin przeprowadź symulację użycia tej monety
            1. Do tymczasowej zmiennej ret_tab_tmp zapisz wynik zwrócony przez rekurencyjne wywołanie funkcji, z argumentami coins oraz value – c
            2. Jeśli ret_tab_tmp jest puste (None), tzn. że nie udało się znaleźć rozwiązania, wtedy przejdź po prostu do następnej monety
            3. Jeśli ret_tab_tmp nie jest puste, to dodaj do niej aktualnie rozważaną monetę c
            4. Jeśli ret_tab_coins jest puste, to właśnie udało się znaleźć nowe rozwiązanie i do ret_tab_coins należy zapisać ret_tab_tmp
            5. W przeciwnym razie jeśli długość ret_tab_tmp jest mniejsza niż długość ret_tab_coins, to do ret_tab_coins zapisz ret_tab_tmp, bo właśnie udało się znaleźć lepsze rozwiązanie
    e. Zwróć ret_tab_coins
4. Przetestuj działanie funkcji uruchamiając kod poniżej (ostatnie obliczenia mogą trwać długo):

```
coins = [1,2,5,10,20]
for value in range(30):
    change = coin_change_naive(coins, value)
    print(value, change)
```

Propozycja rozwiązania – pobierz kod

```
def coin_change_naive(coins, value):

    # if the value is 0, we don't have anything to do
    if value == 0:
        return []

    # we will return  ret_tab_coins, currently we don't know if there is a solution,
    # so we are going to return None
    ret_tab_coins = None

    # make simulation for every coin
    for c in coins:

        # to find a solution the coin needs to be smaller or equal to the value
        if value >= c:
            # we recursively check the problem with decreased value
            ret_tab_tmp = coin_change_naive(coins, value - c)
            # if solution for the smaller number was found
            if ret_tab_tmp != None:
                # we add also the currently considered coin
                ret_tab_tmp.append(c)

                # if no solution till now
                if ret_tab_coins == None:
                    # save the current solution
                    ret_tab_coins = ret_tab_tmp
                # else - check if the new solution is better than the old, if yes - take it
                elif len(ret_tab_tmp) < len(ret_tab_coins):
                    ret_tab_coins = ret_tab_tmp

    return ret_tab_coins


coins = [1,2,5,10,20]
for value in range(30):
    change = coin_change_naive(coins, value)
    print(value, change)
```

Pakowanie plecaka – metoda zachłanna (greedy)

Notatka

Metody zachłanne koncentrują się na tym, żeby szybko zwrócić wynik, który autorowi wydaje się intuicyjny i logicznie uzasadniony, ale niestety, te algorytmy nie zawsze będą zwracać najoptymalniejsze rozwiązanie.

Działanie algorytmu jest zależne od konkretnej sytuacji, np. pakując plecak możesz decydować o umieszczeniu w nim najpierw tych rzeczy, które są najdroższe, albo tych, które są najlżejsze, albo tych, które mają największy iloraz wartość/waga. Dzięki logicznym uproszczeniom algorytmy zachłanne działają szybko.

Lab

1. Dana jest kwota pieniężna value oraz lista dostępnych nominałów monet:

```
value = 17
coins = [1,2,5,10,20]
```

2. Ustal, jaka jest najmniejsza liczba monet i jakie to są monety, która pozwala wydać wskazaną kwotę. Niech funkcja, którą napiszesz zwraca listę nominałów tych monet.
3. Wskazówki – teraz użyj metody zachłannej
    a. Naszym pomysłem może być to, że wypłacając kwotę należy zawsze używać największych nominałów i dopiero kiedy pozostała do wypłacenia kwota maleje, zejść do niższych nominałów – to dość logiczne podejście do problemu
    b. Funkcja może przyjmować argumenty coins i value
    c. Przygotuj roboczą zmienną tab_result na przechowywanie listy monet, która zostanie zwrócona, gdy funkcja się skończy
    d. Napisz pętlę, w której będziesz od wartości value odejmować wartość największej monety, która jest jednak mniejsza lub równa value. Każdorazowo po znalezieniu takiej monety dodaj ją do tab_result:
        i. Warunek pętli value > 0
        ii. W pętli wyznacz największy nominał monety mniejszy lub równy value
        iii. Dodaj tą monetę do tab_result i pomniejsz wartość value o wartość tej monety
    e. Zwróć tab_result
4. Przetestuj działanie tej funkcji w ten sam sposób jak w poprzednim lab:

```
coins = [1,2,5,10,20]
for value in range(30):
    change = coin_change_greedy(coins, value)
    print(value, change)
```

Propozycja rozwiązania – pobierz kod

```
def coin_change_greedy(coins, value):

    tab_result = []

    while value > 0:

        highest_coin = 0
        for c in coins:
            if c <= value and c > highest_coin:
                highest_coin = c

        value -= highest_coin
        tab_result.append(highest_coin)

    return tab_result


coins = [1,2,5,10,20]
for value in range(30):
    change = coin_change_greedy(coins, value)
    print(value, change)
```

Pakowanie plecaka – metoda dynamiczna (dynamic programming)

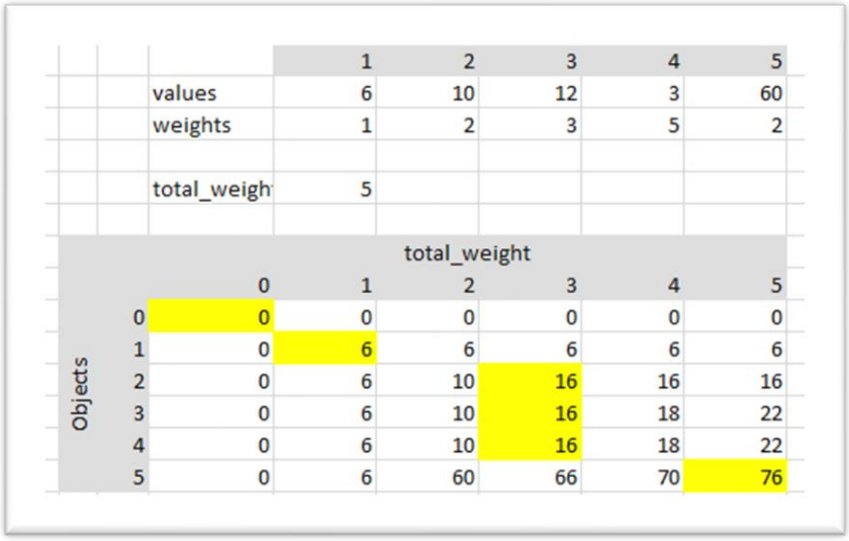
Notatka

Programowanie dynamiczne pozwala na rozwiązywanie problemów optymalizacyjnych, które inaczej byłoby trudno rozwiązać. Polega na podzieleniu skomplikowanego problemu na mniejsze, rozwiązanie ich, a następnie wykorzystanie tych wyników w rozwiązywaniu tych bardziej złożonych problemów.

Dzieje się to przez zbudowanie tabeli, gdzie w wierszach umieszczamy rzeczy, które można zapakować do plecaka. Pierwszy wiersz jest pusty i odpowiada nie umieszczeniu w plecaku niczego, a w kolejnych rzędach umieszcza się informacje o poszczególnych przedmiotach pakowanych do plecaka. W kolumnach umieszcza się limit wagi plecaka w postaci liczb 0, 1,2,3… maksymalna waga plecaka.

Wypełniając tabelkę, przetwarza się ją od górnego lewego rogu, a po zakończeniu przetwarzania wynik będzie w prawym dolnym. Wpisując wartości należy odpowiadać na pytania:

- Czy dany przedmiot zmieści się do danego plecaka? Jeśli tak, to zapamiętać uzyskaną w ten sposób wartość plecaka
- Ile miejsca pozostało w plecaku po ewentualnym umieszczeniu w tym plecaku danej rzeczy? Jeśli jest wolne miejsce, to należy pobrać z wiersza powyżej informacje o rzeczach umieszczanych w plecaku o wielkości odpowiadającej aktualnemu wolnemu miejscu w plecaku. Tą wartość należy dodać do uzyskanej wartości we wcześniejszym kroku
- Jeśli uzyskana wartość jest większa niż wartość w tej samej kolumnie we wcześniejszym wierszu, to należy ją zapisać, jeśli nie, to należy przepisać wartość z wiersza powyżej.



		1	2	3	4	5
	values	6	10	12	3	60
	weights	1	2	3	5	2
	total_weigh	5				

		total_weight					
		0	1	2	3	4	5
Objects	0	0	0	0	0	0	0
	1	0	6	6	6	6	6
	2	0	6	10	16	16	16
	3	0	6	10	16	18	22
	4	0	6	10	16	18	22
	5	0	6	60	66	70	76

Lab

1. Podobnie, jak w poprzednich zadaniach zajmiemy się problemem rozmienienia pieniędzy na monety. Jednak tym razem chcielibyśmy skorzystać a algorytmu dynamicznego.
   Ten przykład jest szczególnie trudny, jeśli chcesz, to przynajmniej za pierwszym razem popatrz na rozwiązanie, postaraj się je zrozumieć, rozrysuj tabelkę, wypełnij ją ręcznie, a potem ewentualnie spróbuj je napisać samodzielnie.

2. Wskazówki:
    a. Uwaga:
        i. W przeciwieństwie do algorytmu plecakowego zależy nam na tym, aby liczba była najmniejsza, zmieniają się więc warunki logiczne.
        ii. Jeśli zdecydujemy się wypłacić kwotę z użyciem w tej chwili analizowanej monety, to resztę monet weźmiemy z tego samego wiersza, ale z kolumny wcześniejszej o wartość analizowanej teraz monety
    b. Funkcja może przyjmować argumenty: coins i value
    c. Utwórz dwie pomocnicze tabele:
        i. tab_numbr będzie to tablica z taką liczbą wierszy, jak liczba monet plus 1 i z taką liczbą kolumn jak value plus 1 (owe +1 to dodatkowy wiersz i kolumna). Domyślnie wpisz do komórek tej tablicy wartość „nieskończoność" – float('inf'), tylko pierwszy wiersz i kolumna mają mieć wartości 0
        ii. tab_coins – będzie to tablica o takim samym rozmiarze jak tab_numbr zawierająca zestawy monet do wypłacenia określonej kwoty. Domyślnie wypełnij ją pustymi listami
    d. Napisz pętlę przetwarzającą każdy wiersz. coin_row_idx wskazujące na numer wiersza powinno zmieniać się od 1 (pomijamy wiersz zerowy) do len(coins)+1
        i. Zapamiętaj przetwarzaną monetę w zmiennej coin (pamiętaj, że sięgając po jej wartość do listy coins należy użyć indeksu o 1 mniejszego)
        ii. Napisz pętlę przetwarzającą każdą kolumnę. Tu też zacznij od 1 i skończ na value+1
            1. Jeśli coin jest większe niż wartość kolumny, to po prostu przepisz wartości z poprzedniego wiersza i tej samej kolumny
            2. W przeciwnym razie:
                a. Jeśli wartość komórki w poprzednim wierszu jest mniejsza niż 1 + wartość w bieżącym wierszu w kolumnie wcześniejszej o wartość monety, to również przepisz wartości z poprzedniego wiersza
                b. W przeciwnym razie (wartość komórki w poprzednim wierszu jest większa niż 1 + wartość w bieżącym wierszu w kolumnie wcześniejsze o wartość monety), to skopiuj do bieżącej komórki wartość z tego samego wiersza z kolumny wcześniejszej o wartość monety i dodaj 1

        (Kopiując wartości komórek, lub dodając 1 aktualizuj też tablicę tab_coins kopiując ją z innej komórki lub dodając do niej nowy element – coin)
    e. Zwróć wartość ostatniej komórki (ostatni wiersz, ostatnia kolumna)
3. Przetestuj działanie funkcji, tak jak w poprzednich LABach. Jeśli chcesz, wyświetl tabelę tab_numbr wiersz po wierszu np. dla wartości kwoty =7. Możesz wtedy prześledzić logikę wypełniania tabeli

```
coins = [1,2,5,10,20]
for value in range(30):
    change = coin_change_dynamic(coins, value)
    print(value, change)
```

Propozycja rozwiązania – pobierz kod

```
def coin_change_dynamic(coins, value):

    tab_numbr = [[ 0  for v in range(value + 1)] for c in range(len(coins) + 1)]
    tab_coins = [[ [] for v in range(value + 1)] for c in range(len(coins) + 1)]
    for i in range(1, value + 1):
        tab_numbr[0][i] = float('inf')  # by default change is impossible

    # for each row of the constructed table
    for coin_row_idx in range(1, len(coins)+1):
        current_coin = coins[coin_row_idx - 1]
        for value_limited in range(1, value + 1):

            # current_coin is too big - we need to take value from the previous row
            if current_coin > value_limited:
                tab_numbr[coin_row_idx][value_limited] = tab_numbr[coin_row_idx -
1][value_limited]
                tab_coins[coin_row_idx][value_limited] = tab_coins[coin_row_idx-
1][value_limited].copy()

            # yes - we could take this coin
            else:
                # the value after adding this coin will be worse (higher) than in the previous
row
                # so copy data from the previous row
                if tab_numbr[coin_row_idx - 1][value_limited] <= 1 +
tab_numbr[coin_row_idx][value_limited - current_coin]:
                    tab_numbr[coin_row_idx][value_limited] = tab_numbr[coin_row_idx -
1][value_limited]
                    tab_coins[coin_row_idx][value_limited] = tab_coins[coin_row_idx -
1][value_limited].copy()
                # the value after adding this coin will be better (lower) than in the previus row
                # so take this coin
                else:
                    tab_numbr[coin_row_idx][value_limited] =  1 +
tab_numbr[coin_row_idx][value_limited - current_coin]
                    tab_coins[coin_row_idx][value_limited] =
tab_coins[coin_row_idx][value_limited - current_coin].copy()
                    tab_coins[coin_row_idx][value_limited].append(current_coin)

#    for row in tab_numbr:
#        print(row)


    return tab_numbr[-1][-1], tab_coins[-1][-1]


coins = [1,2,5,10,20]
value = 7
change = coin_change_dynamic(coins, value)
print(value, change)

# coins = [1,2,5,10,20]
# for value in range(30):
#     change = coin_change_dynamic(coins, value)
#     print(value, change)
```

Ta strona celowo jest pusta

Techniki przyśpieszenia programu: memorization & tabulation

Notatka

Memorization i tabulation mogą przyśpieszyć działanie programu rekurencyjnego, który mógłby być uruchamiany wielokrotnie z tymi samymi argumentami. Obie techniki próbują uniknąć zasobożernej operacji wyznaczania wartości, która już raz była wyznaczana:

- Memorization stosuje pomocniczą tabelę, w której są zapisywane wyniki funkcji uzyskiwane przy określonych wartościach. Jedną z pierwszych instrukcji funkcji wywoływanej rekurencyjnie powinno być sprawdzenie, czy w tej tablicy jest już wynik odpowiadający przekazanym argumentom. Jeśli tak, to należy go zwrócić, a jeśli nie, to trzeba kontynuować obliczanie wartości funkcji. Ostatnim krokiem przed zwróceniem wartości powinno być zapisanie uzyskanego w funkcji do tablicy wyników. Zostaną one ponownie wykorzystane, jeśli ta funkcja zostanie kolejny raz uruchomiona z tymi samymi argumentami
- Tabulation próbuje zupełnie wykluczyć potrzebę wywołania rekurencyjnego funkcji. Zamiast tego próbuje wyznaczać wartość funkcji zaczynając od najmniejszych dopuszczalnych wartości argumentów. Korzystając z pętli zwiększa wartości argumentów i wyznacza kolejne wartości korzystając z tych już wcześniej wyliczonych

Laboratorium

1. W tym zadaniu spróbujemy rozwiązać dylemat francuskiego potentata w produkcji ~~śmierdzących~~ dojrzewających serów pana Pierre au Fromage. Pan Pierre układa sery na desce i zostawia je aż dojrzeją. Każdy kawałek sera ma swój rozmiar zapisany w tablicy table:

```
table = [1,8,4,6,7,5,2,4,1,8,4,6,7,5,2,4,1,8,4,6,7,5]
```

   Codziennie z deski zdejmuje się tylko jeden ser i można go zdjąć wyłącznie z lewej lub prawej strony. Na dodatek zdaniem Pierra ser jest tym lepszy im dłużej dojrzewał, dlatego do wyznaczenia ceny kawałka sera używa się wzoru:
   cena_sera = liczba_dni_dojrzewania * waga_sera
2. Firmowy programista opracował program pozwalający ustalić kolejność w jakiej należy zdejmować sery z deski, żeby uzyskać najlepszą cenę sprzedaży. Oto kod tego programu:

```
def get_cheese(table, start, end, day) :
    tab_result = []

    n = len(table[start:end])
    if n == 1 :
        return day * table[start], table[start:end]

    left, left_tab = get_cheese(table,start+1,end, day+1)
    left += day*table[start]
    right, right_tab = get_cheese(table, start, end-1, day+1)
    right+= day*table[end-1]

    if left >= right:
        tab_result.append(table[start])
        tab_result.extend(left_tab)
        return left, tab_result
    else:
        tab_result.append(table[end-1])
        tab_result.extend(right_tab)
        return right, tab_result
```

```
table = [1,8,4,6,7,5,2,4,1,8,4,6,7,5,2,4,1,8,4,6,7,5]
print(get_cheese(table,0,len(table), 1))
```

3. Przeanalizuj ten program. Bardzo ogólnie:
    a. Funkcję wywołujemy z parametrami wskazującymi na deskę serów (table), pozycję od której i do której należy analizować tę deskę (s i e) oraz numerem dnia, kiedy symulujemy zdjęcie sera z deski
    b. Jeśli na desce został tylko jeden kawałek sera, to należy go zdjąć i zwrócić informację o jego cenie wraz z jednoelementową listą, która zachowuje informację o kolejności ściągania serów z deski
    c. Jeśli na desce jest więcej serów, to należy rozważyć 2 przypadki:
        i. Ściągnięcie sera po lewej
        ii. Ściągnięcie sera po prawej
    d. Potem należy zwrócić informację o kolejności ściągania serów, która daje lepszy końcowy dochód
4. Niestety, uruchomienie programu zajmuje trochę dużo czasu (u mnie ok. 15 sekund).
5. Korzystając z techniki memoryzacji, spraw, aby funkcja działała lepiej
6. Wskazówka:
    a. Utwórz tablicę do zapamiętywania pośrednich wyników indeksowaną parametrami funkcji: day, start i end:

```
mem = [[[None for day in range(len(table))] for s in range(len(table)) ] for e in
range(len(table))]
```

    b. Wywołując funkcję przekazuj jako ostatni parametr tą tablicę
    c. Na początku sprawdzaj, czy w tablicy jest już wcześniej wyznaczona niepusta wartość (uwaga na indeksację – dzień liczymy od 1, a tablicę indeksujemy od 0, podobnie wartość argumentu e oznacza ostatni element listy, ale przetwarzanie danych kończy się na e-1)
    d. Jeśli taką wartość udało się znaleźć, to ją zwróć
    e. Jeśli wartości nie ma jeszcze w tablicy, to ją wyznacz, a następnie zapisz w tablicy do wykorzystania w przyszłości

Propozycja rozwiązania – pobierz kod

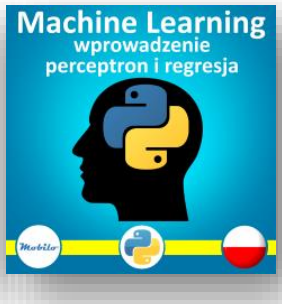
```
def get_cheese_mem(table,start,end, day, mem):

    if mem[day-1][start][end-1] != None:
        return mem[day-1][start][end-1]

    tab_result = []

    n = len(table[start:end])
    if n == 1 :
        mem[day-1][start][end-1] = [day * table[start], table[start:end]]
        return [day * table[start], table[start:end]]

    left, left_tab = get_cheese_mem(table,start+1,end, day+1, mem)
    left += day*table[start]
    right, right_tab = get_cheese_mem(table, start, end-1, day+1, mem)
    right+= day*table[end-1]

    if left >= right:
        tab_result.append(table[start])
        tab_result.extend(left_tab)
        mem[day-1][start][end-1] = left, tab_result
        return left, tab_result

    else:
        tab_result.append(table[end-1])
        tab_result.extend(right_tab)
        mem[day-1][start][end-1] = right, tab_result
        return right, tab_result


table = [1,8,4,6,7,5,2,4,1,8,4,6,7,5,2,4,1,8,4,6,7,5]
mem = [[[None for day in range(len(table))] for s in range(len(table)) ] for e in
range(len(table))]
print(get_cheese_mem(table,0,len(table), 1, mem))
```

Spróbuj też!


Python
Dla początkujących


Analiza danych
w Python i PANDAS


Python dla średnio
zaawansowanych


Machine Learning
wprowadzenie
perceptron i regresja


Python Flask
Aplikacje webowe


Myślenie algorytmiczne
Python - struktury i algorytmy,
które musisz znać


Myślenie algorytmiczne
Grafy w Pythonie
od podstaw


Linux
dla początkujących


Administrator Linux
Instalacja i konfiguracja


Administrator Linux
Skryptowanie w BASH


PostgreSQL
dla początkujących


Microsoft Azure
AZ-900


Azure Administrator
Storage (AZ-104)


PowerShell
dla administratora Windows


SQL Server #1
Instalacja i narzędzia
Microsoft
SQL Server


SQL Server #2
Zapytania SQL
Microsoft
SQL Server


SQL Server #3
Programowanie
typy zaawansowane, XML
Microsoft
SQL Server


SQL Server #4
Modyfikacja danych
transakcje, obsługa błędów
Microsoft
SQL Server


MS Excel Tips & Tricks


Język R
dla początkujących


Reporting Services
Tworzenie raportów
Microsoft
SQL Server


SAP ASE 16 / Sybase
Installation and Configuration