# Programowanie
## *po 5 wykładzie*

Andrzej Giniewicz

3.04.2024

W tej części materiałów zajmiemy się rekurencyjnymi algorytmami sortowania.

## 1 Sortowanie przez scalanie

Pomysł na sortowanie przez scalanie jest stosunkowo prosty — łatwo połączyć dwie posortowane listy w jedną. Z każdej z nich wybieramy mniejszy element z początku, aż jedna się wyczerpie. Wtedy tę, która pozostała, doczepiamy na koniec wyniku. Co więcej, wiemy, że każda lista długości 0 lub 1 jest zawsze posortowana. Wobec tego, gdy do posortowania mamy listę długości $n$, możemy podzielić ją na dwie listy długości około $\frac{n}{2}$, wywołać funkcję rekurencyjnie, aby posortowała nam połówki i potem połączyć dwie listy w jedną.

Spróbujmy zaimplementować najpierw funkcję łączącą dwie posortowane listy w jedną nową. Załóżmy, że pierwsza lista powstała z pierwszej połowy listy wejściowej. Skorzystamy z tego, aby w przypadku remisów, najpierw „zdjąć" wartości z lewej listy a dopiero potem z prawej — dzięki temu nasz algorytm będzie stabilny. Dla wydajności nie będziemy modyfikować list wejściowych, tylko przejdziemy po odpowiednich wartościach, korzystając z dwóch indeksów przechodzących po oryginalnych listach i jednym, przechodzącym po wyniku.

Przykładowy kod został przeniesiony na nową stronę, aby nie dzielić go w połowie, co utrudniłoby jego zrozumienie. Z tego powodu, w tekście zaczynamy analizą, ale odnosi się ona do programu znajdującego się na kolejnej stronie.

Zaczynamy od zmierzenia list wejściowych i stworzenia nowej listy na rezultaty — po zmierzeniu listy `lewa` i `prawa` wiemy, jaką długość będzie miał rezultat. Ponieważ nie ma możliwości stworzenia pustej listy konkretnej długości, tworzymy listę zer, które potem zastąpimy właściwymi elementami z lewej lub prawej listy. Zmienna `indeks_lewy` wskazuje na pierwszy element listy `lewa`, zmienna `indeks_prawy` wskazuje na pierwszy element listy `prawa`, a zmienna `indeks_wynik` wskazuje na pierwszy element zmiennej `wynik`. Nasz plan działania ma dwa etapy. W pierwszym etapie porównujemy dwie wartości wskazywane przez zmienne `indeks_lewy` i `indeks_prawy`. Wybieramy mniejszą z wartości i jeśli mniejsza (lub równa) jest wartość w liście `lewa`, przepisujemy element wskazywany przez `indeks_lewy` pod zmienną wskazywaną przez `indeks_wynik`, następnie zwiększamy oba te indeksy o jeden. Gdy mniejszy jest element z prawej listy, postępujemy analogicznie. Etap ten trwa tak długo,

jak obie listy mają elementy, które nie zostały przepisane do wyniku. Po przepisaniu do wyniku jednej z listy zaczynamy drugi etap. Teraz musimy przepisać pozostałe elementy z obu list. Robią to dwie kolejne pętle. Tym razem już nie muszą sprawdzać, który element jest mniejszy, bo wiemy, że obie listy były posortowane — wobec tego, możemy bezmyślnie przepisywać je tak długo, aż skończą się pozostałe elementy.

```
def scal(lewa, prawa, relacja):
    n_lewy = len(lewa)
    n_prawy = len(prawa)
    wynik = [0]*(n_lewy+n_prawy)
    indeks_lewy = 0
    indeks_prawy = 0
    indeks_wynik = 0
    while indeks_lewy < n_lewy and indeks_prawy < n_prawy:
        if relacja(lewa[indeks_lewy], prawa[indeks_prawy]):
            wynik[indeks_wynik] = lewa[indeks_lewy]
            indeks_lewy += 1
        else:
            wynik[indeks_wynik] = prawa[indeks_prawy]
            indeks_prawy += 1
        indeks_wynik += 1
    while indeks_lewy < n_lewy:
        wynik[indeks_wynik] = lewa[indeks_lewy]
        indeks_lewy += 1
        indeks_wynik += 1
    while indeks_prawy < n_prawy:
        wynik[indeks_wynik] = prawa[indeks_prawy]
        indeks_prawy += 1
        indeks_wynik += 1
    return wynik
```

Zaimplementujmy teraz nasz pomysł na sortowanie, korzystający z funkcji `scal`.

```
def przez_scalanie(lista, relacja=lambda x,y: x <= y):
    n = len(lista)
    if n <= 1:
        return lista.copy()
    k = n//2
    lewa, prawa = lista[:k], lista[k:]
    lewa = przez_scalanie(lewa, relacja)
    prawa = przez_scalanie(prawa, relacja)
    return scal(lewa, prawa, relacja)
```

Ponieważ w większości przypadków, poza listą pustą i długości 1 będziemy używać funkcji `scal`, będziemy wtedy zwracać nową listę, która na pewno nie ma innych odwołań nigdzie w kodzie. Aby zagwarantować, że zachowanie funkcji przy $n \in \{0, 1\}$ jest takie samo, zwracamy w tych przypadkach nie tę samą listę, którą otrzymaliśmy w argumencie, tylko jej kopię.

Algorytm taki działa w czasie proporcjonalnym do $n\log(n)$, w większości sytuacji jest więc szybszy, niż te omawiane na wykładzie, które w najgorszym przypadku miały czas działania proporcjonalny do funkcji kwadratowej długości listy. W Internecie znajdują się wizualizacje tego oraz innych algorytmów. Zachęcam, aby je obejrzeć, ponieważ pomagają zrozumieć działanie kodu[1].

Fakt, że sortowanie przez scalanie jest szybkie, nie oznacza, że nie da się go poprawić. Wykonując pomiar czasu, możemy się przekonać, że dla krótkich list długości co najwyżej 8, sortowanie przez wstawianie jest szybsze. Wobec tego wykorzystując je w warunku początkowym rekurencji, uzyskamy algorytm znacznie szybszy.

```
def wstawianie(lista, relacja):
    for i in range(1, len(lista)):
        L_i = lista[i]
        j = i
        while j>0 and not relacja(lista[j-1], L_i):
            lista[j] = lista[j-1]
            j -= 1
        lista[j] = L_i

def przez_scalanie(lista, relacja=lambda x,y: x <= y):
    n = len(lista)
    if n <= 8:
        lista = lista.copy()
        wstawianie(lista, relacja)
        return lista
    k = n//2
    lewa, prawa = lista[:k], lista[k:]
    lewa = przez_scalanie(lewa, relacja)
    prawa = przez_scalanie(prawa, relacja)
    return scal(lewa, prawa, relacja)
```

Przy takiej optymalizacji, średni czas sortowania listy długości 2000 spada z 9.3*ms* do 6.5*ms*, czyli około 1.4 raza szybciej. Jeśli do optymalizacji tego typu, czyli połączenia algorytmu

[1] Taneczne wykonania opisywanych algorytmów możemy znaleźć wśród filmów grupy AlgoRythmics. Możemy zobaczyć sortowanie bąbelkowe (`https://youtu.be/lyZQPjUT5B4`), sortowanie przez wybieranie (`https://youtu.be/Ns4TPTC8whw`), sortowanie przez wstawianie (`https://youtu.be/ROalU379l3U`) oraz sortowanie przez scalanie (`https://youtu.be/XaqR3G_NVoo`). Mniej taneczne wykonanie wszystkich omawianych algorytmów znajdziemy na przykład na `https://youtu.be/INHF_5RIxTE`, natomiast wyścigi algorytmów (tym razem bez aktorów) możemy zobaczyć na stronie `https://www.toptal.com/developers/sorting-algorithms`.

sortowania przez scalanie i wstawianie, dołączymy bardziej zaawansowaną technikę zwaną galopem, która pozwala na pomijanie niektórych porównań dla serii posortowanych wartości, uzyskamy jeszcze lepsze rozwiązanie — algorytm o nazwie TimSort[2]. TimSort jest jednym z najszybszych ogólnych algorytmów sortowania, dostępnym między innymi w językach Python, Java, Rust czy Swift jako domyślna implementacja metod sortujących listy w oparciu o porównania. W momencie, gdy piszemy `lista.sort()` lub `sorted(lista)`, to właśnie z tego algorytmu korzysta Python.

Istnieje wiele innych algorytmów sortujących, między innymi QuickSort, ShellSort lub HeapSort należą do klasyki informatyki. Istnieje też dużo nowocześniejszych algorytmów. Ostatecznie, w praktyce zwykle korzystamy z gotowej implementacji sortowania, co zmniejsza szansę na pomyłkę. Domyślne algorytmy są zawsze bardzo szybkie i przetestowane przez miliony ludzi. Niemniej jednak wciąż opłaca się dowiedzieć, co tak naprawdę dzieje się pod maską, gdy uruchamiamy na pozór prostą metodę, dopisując za listą niewinnie wyglądające `.sort()`. W napisie tym kryje się dużo pracy, zarazem projektantów algorytmu, jego programistów oraz komputera. W wersji minimum musimy być świadomi tej ostatniej, ponieważ pozornie krótka linijka, może trwać znaczącą ilość czasu i stanowić ukryty koszt, który łatwo pominąć w analizie czasu działania kodu.

## 2 Opcje standardowego sortowania

Zarazem metoda `sort`, jak i funkcja `sorted`, przyjmują dwa opcjonalne parametry, `key` oraz `reverse`. Funkcje te pozwalają określić relację porównywania, ale w nieco inny sposób, niż nasze implementacje podczas zajęć. W parametrze `key` przekazujemy funkcję, która przekształca elementy listy na elementy porównywane, natomiast w `reverse` podajemy wartość logiczną mówiącą, czy zamienić kolejność sortowania. Jeśli mamy funkcję `key` i stałą `reverse`, można zdefiniować relację.

```
def relacja(key=lambda element: element, reverse=False):
    if reverse:
        return lambda x, y: key(y) <= key(x)
    else:
        return lambda x, y: key(x) <= key(y)
```

W jaki sposób podawać klucze? Stwórzmy klucz pozwalający posortować trójki z imieniem, nazwiskiem i wynikiem punktowym, takie jak w przykładzie z wcześniejszych notatek, w kolejności nazwiska a potem imienia. Możemy to zrobić, pisząc

```
lista.sort(key=lambda krotka: (krotka[1], krotka[0]))
```

[2]TimSort w przeciwieństwie do wielu algorytmów sortowania nie powstał w środowisku naukowym, tylko w bardzo praktycznym — pierwszy raz został opisany na liście mailingowej Pythona w wiadomości proponującej zmianę wcześniejszego algorytmu sortowania na nowy, dający bardzo wydaje rezultaty. Badania i analiza algorytmu miała miejscec później. Jeden z artykułów dotyczących algorytmu TimSort to „Merge Strategies: from Merge Sort to TimSort" autorstwa Nicolasa Augera, Cyrila Nicauda, Carine Pivoteau dostępny na `https://hal-upec-upem.archives-ouvertes.fr/hal-01212839v2/document`.

Dzięki takiemu zabiegowi, Python podczas sortowania, ale tylko na potrzeby sortowania, zamieni krotki trójelementowe postaci (`imię, nazwisko, punkty`) na krotki (`nazwisko, imię`). Przez to standardowe sortowanie leksykograficzne krotek uporządkuje nam wartości w pożądanej kolejności, najpierw po nazwisku, a gdy wartości te są równe, po imieniu, bez zwracania uwagi na wynik punktowy (traktując wszystkie osoby o tym samym nazwisku i imieniu jako jedną klasę równoważności).

Jeśli chcielibyśmy zobaczyć te dane w odwrotnej kolejności, powinniśmy napisać

```
lista.sort(key=lambda krotka: (krotka[1], krotka[0]), reverse=True)
```

co zagwarantuje nam odwrócenie relacji a tym samym odwrócenie kolejności na liście z rosnącej na malejącą.