Programowanie
przed 5 wykładem

Andrzej Giniewicz

05.04.2024

Dzisiejszy blok tematyczny będzie poświęcony sortowaniu.

1 Sortowanie elementów listy

W języku Python mamy dostępne dwa sposoby na posortowanie listy — jeden modyfikujący istniejącą listę

```
lista.sort()
```

oraz drugi, zwracający nową posortowaną listę na podstawie istniejącej

```
nowa_lista = sorted(lista)
```

Oba są często używane, ponieważ posortowanie danych często jest wstępnym krokiem dla wielu bardziej złożonych algorytmów. W niniejszej części przyjrzymy się bliżej tej operacji, zanim jednak poznamy algorytm stosowany w Pythonie, prześledzimy kilka prostszych algorytmów.

2 Podział algorytmów sortowania

Algorytmy sortowania możemy podzielić na kilka sposobów:

1. ze względu na to, czy są oparte na porównaniach wartości pomiędzy dwoma elementami za pomocą praporządku spójnego (relacji zwrotnej, przechodniej i spójnej — więcej o tym za chwilę), czy na innym pomyśle na porządkowanie,
2. ze względu na szybkość działania w zależności od długości listy (między innymi działające w czasie liniowym, quasi-liniowym lub kwadratowym — więcej o tym za chwilę),

3. ze względu na to, czy są stabilne (kolejność elementów w jednej klasie równoważności w liście posortowanej i nieposortowanej jest zawsze taka sama — więcej o tym za chwilę) lub nie są stabilne (kolejność elementów w jednej klasie równoważności w liście posortowanej i nieposortowanej może, choć nie musi się różnić),

4. ze względu na to, czy działają w miejscu (modyfikują listę, zmieniając kolejność elementów i potrzebują co najwyżej stałej ilości dodatkowej pamięci do wykonania zadania sortowania) oraz te, które nie działają w miejscu (potrzebują dodatkowej pamięci zależnej od liczby elementów listy, na przykład tworzą jej kopię),

5. ze względu na sposób implementacji, czyli takie, które są zaimplementowane w sposób iteracyjny lub rekurencyjny.

3 Sortowanie oparte na porównaniach

Przyjrzyjmy się pierwszemu kryterium. Algorytm jest oparty na porównaniach, jeśli jedyne co zakłada to fakt, że dwa elementy listy można porównać za pomocą operatora określającego na zbiorze wartości listy praporządek spójny. Relacja $\preceq$ na zbiorze wartości listy $\mathcal{X}$ jest praporządkiem spójnym, jeśli jest:

- zwrotna, czyli dla każdego elementu $a \in \mathcal{X}$ zachodzi $a \preceq a$,
- przechodnia, czyli dla każdej trójki elementów $a, b, c \in \mathcal{X}$ zachodzenie relacji $a \preceq b$ oraz $b \preceq c$ pociąga za sobą zachodzenie relacji $a \preceq c$,
- spójna, czyli dla każdej pary elementów $a, b \in x$, $a \neq b$ zachodzi $a \preceq b$ lub $b \preceq a$.

Spójność relacji gwarantuje, że każde dwa elementy można ze sobą porównać. Przechodniość jest konieczna, aby stwierdzić, że cała lista jest posortowana na podstawie porównywania jedynie elementów parami. Zwrotność gwarantuje nam, że algorytm będzie działał, gdy na liście znajdować się będą duplikaty, czyli elementy o takiej samej wartości. Przy rezygnacji z któregokolwiek z trzech powyższych założeń na temat relacji, mogłoby się okazać, że dla niektórych list nie jesteśmy w stanie określić, czy są posortowane, czy nie — a co za tym idzie, nie potrafilibyśmy posortować permutacji tych list.

Niekiedy można spotkać inną definicję spójności, która zawiera w sobie zwrotność. Wtedy warunki byłyby tylko dwa — relacja musiałaby być:

- przechodnia, czyli dla każdej trójki elementów $a, b, c \in \mathcal{X}$ zachodzenie relacji $a \preceq b$ oraz $b \preceq c$ pociąga za sobą zachodzenie relacji $a \preceq c$,
- spójna, czyli dla każdej pary elementów $a, b \in x$, zachodzi $a \preceq b$ lub $b \preceq a$.

Warto zwrócić uwagę, że praporządek nie musi być antysymetryczny. Relacja $\preceq$ na zbiorze wartości listy $\mathcal{X}$ jest antysymetryczna, jeśli dla każdej pary elementów $a, b \in \mathcal{X}$ zachodzenie relacji $a \preceq b$ oraz $b \preceq a$ jednocześnie pociąga za sobą $a = b$, innymi słowy $a \preceq b \preceq a \implies a = b$. Praporządki, które są dodatkowo antysymetryczne, nazywamy słabymi porządkami częściowymi, natomiast praporządki spójne, które są dodatkowo

antysymetryczne, nazywamy słabymi porządkami liniowymi. Przykładem słabego porządku liniowego jest relacja $\leqslant$ na liczbach rzeczywistych lub ich podzbiorach lub porządek leksykograficzny na napisach. Relacje te są słabymi porządkami liniowymi, więc są również praporządkami spójnymi, czyli mogą być używane w algorytmach sortowania. Jako przykład praporządku spójnego, który nie jest słabym porządkiem liniowym, możemy określić sortowanie listy krotek postaci `(imię, nazwisko, wynik_egzaminu)` według nazwiska i imienia. Może się zdarzyć, że na liście będą dwie osoby o tym samym imieniu i nazwisku, ale różnych wynikach egzaminu. Weźmy dwa elementy `("Jan", "Kowalski", 75)` oraz `("Jan", "Kowalski", 80)`. Dla relacji porównującej nazwisko i imię zachodzi jednocześnie

$$\texttt{("Jan", "Kowalski", 75)} \preceq \texttt{("Jan", "Kowalski", 80)}$$

oraz

$$\texttt{("Jan", "Kowalski", 80)} \preceq \texttt{("Jan", "Kowalski", 75)}$$

Gdyby wymusić, że sortować możemy tylko za pomocą relacji antysymetrycznych, oznaczałoby to (z definicji relacji antysymetrycznych), że

$$\texttt{("Jan", "Kowalski", 80)} = \texttt{("Jan", "Kowalski", 75)},$$

co oczywiście nie jest prawdą, ponieważ są to dwie różne osoby. Wobec tego nie moglibyśmy posortować takiej listy trójek jedynie po nazwiskach i imionach, co jest ograniczające.

4 Stabilność sortowania

Aby określić stabilność sortowania, musimy określić relację równoważności mówiącą, kiedy elementy są tożsame. Jeśli stosujemy sortowanie oparte o porównania i dysponujemy praporządkiem spójnym $\preceq$, zawsze możemy określić relację równoważności $a \equiv b$, która zachodzi, jeśli jednocześnie są spełnione $a \preceq b$ oraz $b \preceq a$. Jeśli sortowanie nie jest oparte o porównywanie, musimy określić relację równoważności inaczej — w najprostszym przypadku, każdy zbiór elementów może mieć określoną relację „ten sam" lub „taki sam".

Dla każdego elementu $a \in \mathcal{X}$ możemy zdefiniować jego klasę równoważności jako zbiór wszystkich elementów $b \in \mathcal{X}$ spełniających $a \equiv b$. Sortowanie jest stabilne, jeśli nie zmienia kolejności elementów w jednej klasie równoważności.

W przypadku powyżej `("Jan", "Kowalski", 80)` oraz `("Jan", "Kowalski", 75)` nie są równe, ale są w jednej klasie równoważności, jeśli stosujemy relację porządkującą po nazwiskach i imionach, wobec tego algorytm stabilny musi zagwarantować, że jeśli krotka `("Jan", "Kowalski", 80)` występowała w nieposortowanej liście wcześniej niż krotka `("Jan", "Kowalski", 75)`, musi również wystąpić wcześniej w liście posortowanej.

5 Szybkość sortowania

Bez trudu można znaleźć algorytm, który sortuje wartości w czasie proporcjonalnym do n^2, gdzie n to liczba elementów listy, choćby porównując każdy element z każdym.

Dla algorytmów opartych o porównania, przy odrobinie gimnastyki znajdziemy algorytm działający w czasie proporcjonalnym do $n\log(n)$. Niestety szybszych algorytmów nie odnajdziemy bez dodatkowych założeń, samo istnienie relacji praporządku zupełnego i klas równoważności elementów nie wystarcza.

Jeśli założymy, że w liście wejściowej znajduje się tylko k różnych klas jednoznaczności, możemy przejść po liście i dodać elementy oryginalnej listy do jednej z k list odpowiadających klasom. Wybór właściwej listy będzie trwał czas proporcjonalny do $\log(k)$, więc powrzucanie wszystkich elementów do jednej z k klas trwa czas proporcjonalny do $n\log(k)$. Następnie, mając k różnych klas równoważności, musimy je posortować, korzystając z innego algorytmu sortowania, co wobec tego potrwa czas proporcjonalny do $k\log(k)$. W sumie te dwie operacje potrwają $(n+k)\log(k)$. W ostatnim kroku musimy scalić elementy z k „worków" w jedną listę, co może potrwać czas proporcjonalny do $n+k$, ostatecznie dając wynikowy czas proporcjonalny do $(n+k)(\log(k)+1)$. Jeśli k jest dużo mniejsze od n, zastosowanie takiego algorytmu będzie bardziej wydajne, niż algorytmu który jest proporcjonalny do $n\log(n)$. Niestety, jeśli k okaże się bliskie n, takie rozwiązanie może być nawet dwukrotnie wolniejsze niż zwykłe sortowanie.

Jeśli dalej założymy, że lista k klas równoważności jest z góry znana i może być utożsamiona z liczbami naturalnymi od 0 do $k-1$, można przygotować jeszcze szybszy algorytm. Jeśli wyniki to lista wyników rzutów kostką, czyli liczby od 1 do 6, właściwy „worek" na elementy znajdziemy w czasie stałym, odejmując 1 od wyniku rzutu kostką, a nie $\log(k)$. W efekcie czas całego sortowania będzie proporcjonalny do n, a nie $n\log(n)$.

Algorytmy takie, które mogą działać wydajnie dla małej liczby klas równoważności, nazywamy algorytmami działającymi w czasie liniowym, ponieważ czas ich działania jest proporcjonalny do liniowej funkcji rozmiaru listy. Niemniej jednak takie algorytmy, choć są szybkie, wymagają implementacji dobranej do konkretnej sytuacji — funkcja nie będzie uniwersalna, tylko będzie działać jedynie dla list o małej a czasem nawet konkretnej liczbie klas równoważności. My będziemy skupiać się na algorytmach sortowania, które nie mają dodatkowych założeń o zawartości listy oprócz tego, że możemy je porównywać ze sobą, więc najlepszym czasem, jaki możemy uzyskać, będzie czas w przybliżeniu proporcjonalny do $n\log(n)$ (będziemy mówić, że to algorytmy działające w czasie quasi-liniowym), natomiast najwolniejszymi, jakim będziemy zainteresowani, są algorytmy o czasie w przybliżeniu proporcjonalnym do n^2 (będziemy mówić, że to algorytmy działające w czasie kwadratowym).

6 Podsumowanie

Jeśli zdarza Ci się grywać w karty, przed wykładem spróbuj zastanowić się, w jaki sposób sortujesz karty na dłoni tak, aby podzielone były kolorami i poukładane wartościami od najniższej do najwyższej. Spróbujemy wspólnie zaimplementować takie pomysły na sortowanie listy, które się na tym opierają i spełniają następujące warunki:

1. będą to algorytmy oparte o porównania,

2. będą działać dla dowolnego praporządku zupełnego przezkazanego w formie argumentu listy,
3. będą w najgorszym przypadku miały czas kwadratowy,
4. będą działać w miejscu,
5. będą zaimplementowane iteracyjnie,
6. ich implementacja będzie względnie prosta (zajmą mniej niż 10 linijek kodu).

Aby przygotować się dobrze do zajęć, spróbujcie zaimplementować w Pythonie swoje pomysły na sortowanie kart. Możecie wykorzystać szablon kodu poniżej:

```
def moje_sortowanie(lista, relacja=lambda x, y: x <= y):
    ...
```

Wewnątrz kodu możesz używać funkcji `relacja(x, y)`, która zwraca `True`, jeśli zachodzi relacja $x \preceq y$. Wartość domyślna jest dobrana tak, aby algorytm działał z typami Pythona takimi jak liczby, krotki oraz napisy.