11.04.2018, Wrocław

**Analiza algorytmów sortowania**

Daniel Eryk Wojciechowski

Nr indeksu: 242395

Algorytmy i Struktury Danych – Laboratoria

Studia stacjonarne – Rok 1 – Semestr 2

Środa 9:15

Prowadzący: Mgr Rajmund Klemiński

Repozytorium projektu: <https://github.com/wojciechowski08/sem2-aisd-lab-kleminski-python>

**Algorytmy sortowania**

Tematem pracy jest porównanie wydajności trzech różnych metod sortowania: QuickSort, MergeSort i ShellSort. Dwa z trzech porównywanych algorytmów (QuickSort, MergeSort) charakteryzują się przeciętną złożonością obliczeniową , a kolejny z porównywanych sposobów (ShellSort) - , przy czym stały współczynnik kryjący się za notacją O(·) jest różny dla wszystkich trzech metod. Dla celów poniżej prezentowanych badań jako język implementacji wybrano Python w wersji 3.x, a kompilacji dokonano kompilatorem wbudowanym w Python 3.x pobrany bezpośrednio ze strony python.org, zaś architekturą dla której generowany był kod jest architektura 32bitowa.

Zestawienie

Wybrane algorytmy na tle innych charatkeryzują się wysoką wydajnością pracy dla różnych przypadków

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Lp.** | **Nazwa algorytmu** | **Klasa złożoności** | | | **Stabilność** | **Sortowanie w miejscu** | **Zalecane?** |
| optymistyczna | typowa | pesymistyczna |
| 1. | [Bogo Sort](http://eduinf.waw.pl/inf/alg/003_sort/0002.php) | *O*(1) | *O*(*n×n*!) | *O*(∞) | NIE | TAK | NIE!!! |
| 2. | [Stupid Sort](http://eduinf.waw.pl/inf/alg/003_sort/0003.php) | *O*(*n*)*... O*(*n*2) | *O*(*n*3) | *O*(*n*3) | TAK | TAK | NIE |
| 3. | [Bubble Sort I](http://eduinf.waw.pl/inf/alg/003_sort/0004.php) | *O*(*n*2) | *O*(*n*2) | *O*(*n*2) | TAK | TAK | NIE |
| 4. | [Bubble Sort II](http://eduinf.waw.pl/inf/alg/003_sort/0005.php) | *O*(*n*2) | *O*(*n*2) | *O*(*n*2) | TAK | TAK | NIE |
| 5. | [Bubble Sort III](http://eduinf.waw.pl/inf/alg/003_sort/0006.php) | *O*(*n*)*... O*(*n*2) | *O*(*n*2) | *O*(*n*2) | TAK | TAK | TAK/NIE |
| 6. | [Bubble Sort IV](http://eduinf.waw.pl/inf/alg/003_sort/0007.php) | *O*(*n*) | *O*(*n*2) | *O*(*n*2) | TAK | TAK | TAK |
| 7. | [Bidirectional Bubble Sort](http://eduinf.waw.pl/inf/alg/003_sort/0008.php) | *O*(*n*) | *O*(*n*2) | *O*(*n*2) | TAK | TAK | TAK |
| 8. | [Selection Sort](http://eduinf.waw.pl/inf/alg/003_sort/0009.php) | *O*(*n*2) | *O*(*n*2) | *O*(*n*2) | NIE | TAK | TAK/NIE |
| 9. | [Insertion Sort](http://eduinf.waw.pl/inf/alg/003_sort/0010.php) | *O*(*n*) | *O*(*n*2) | *O*(*n*2) | TAK | TAK | TAK |
| 10. | [Binary Insertion Sort](http://eduinf.waw.pl/inf/alg/003_sort/0011.php) | *O*(*n* log*n*) | *O*(*n*2) | *O*(*n*2) | TAK | TAK | NIE |
| 11. | [Shell Sort](http://eduinf.waw.pl/inf/alg/003_sort/0012.php) | *O*(*n*1,14*)* | *O*(*n*1,15) | *O*(*n*1,15) | NIE | TAK | TAK |
| 12. | [Merge Sort](http://eduinf.waw.pl/inf/alg/003_sort/0013.php) | *O*(*n*log *n*) | *O*(*n*log *n*) | *O*(*n×*log*n*) | TAK | NIE | TAK |
| 13. | [Heap Sort](http://eduinf.waw.pl/inf/alg/003_sort/0017.php) | *O*(*n*log*n*) | *O*(*n*log*n*) | *O*(*n×*log*n*) | NIE | TAK | TAK |
| 14. | [Quick Sort](http://eduinf.waw.pl/inf/alg/003_sort/0018.php) | *O*(*n*log*n*) | *O*(*n*log*n*) | *O*(*n*2) | NIE | TAK | TAK |
| 15. | [Bucket Sort I](http://eduinf.waw.pl/inf/alg/003_sort/0020.php) | *O*(*m* + *n*) | *O*(*m* + *n*) | *O*(*m* + *n*) | NIE | NIE | TAK/NIE |

**Krótki opis algorytmów sortowania**

***QuickSort***

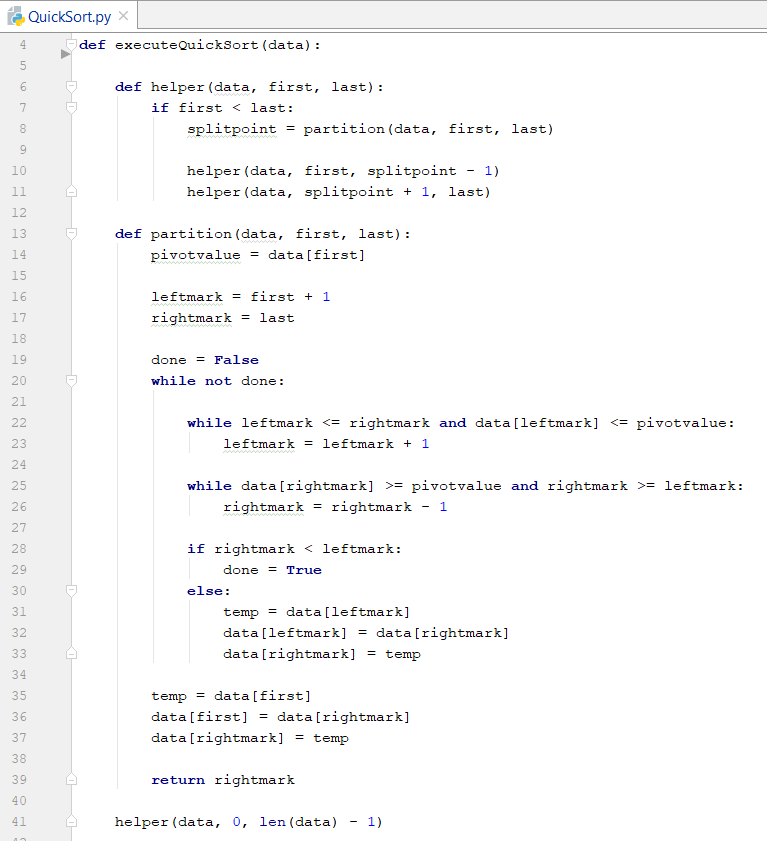
Sortowanie szybkie zostało wynalezione przez angielskiego informatyka, profesora [Tony'ego Hoare'a](http://en.wikipedia.org/wiki/C._A._R._Hoare) w latach 60-tych XX wieku. W przypadku typowym algorytm ten jest najszybszym algorytmem sortującym z klasy złożoności obliczeniowej  - stąd pochodzi jego popularność w zastosowaniach. Musimy jednak pamiętać, iż w pewnych sytuacjach (zależnych od sposobu wyboru piwotu oraz niekorzystnego ułożenia danych wejściowych) klasa złożoności obliczeniowej tego algorytmu może się degradować do , co więcej, poziom wywołań rekurencyjnych może spowodować przepełnienie stosu i zablokowanie komputera. Z tych powodów algorytmu sortowania szybkiego nie można stosować bezmyślnie w każdej sytuacji tylko dlatego, iż jest uważany za jeden z najszybszych algorytmów sortujących - zawsze należy przeprowadzić analizę możliwych danych wejściowych właśnie pod kątem przypadku niekorzystnego.

Algorytm sortowania szybkiego opiera się na strategii "dziel i zwyciężaj" (ang. divide and conquer), którą możemy krótko scharakteryzować w trzech punktach:

* **DZIEL** - problem główny zostaje podzielony na podproblemy
* **ZWYCIĘŻAJ** - znajdujemy rozwiązanie podproblemów
* **POŁĄCZ** - rozwiązania podproblemów zostają połączone w rozwiązanie problemu głównego

Idea sortowania szybkiego jest następująca:

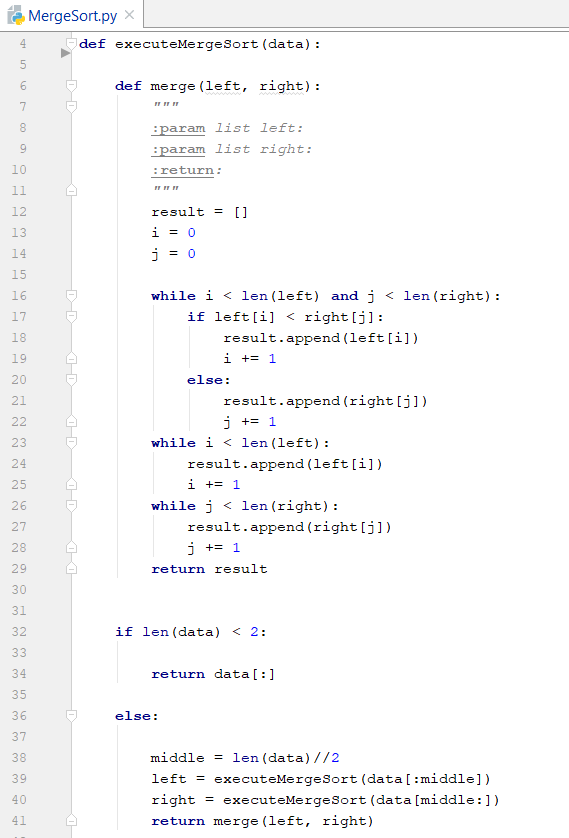
|  |  |
| --- | --- |
| **DZIEL** | najpierw sortowany zbiór dzielimy na dwie części w taki sposób, aby wszystkie elementy leżące w pierwszej części (zwanej lewą partycją) były mniejsze lub równe od wszystkich elementów drugiej części zbioru (zwanej prawą partycją). |
| **ZWYCIĘŻAJ**: | każdą z partycji sortujemy rekurencyjnie tym samym algorytmem. |
| **POŁĄCZ** | połączenie tych dwóch partycji w jeden zbiór daje w wyniku zbiór posortowany. |



***MergeSort***

Wynaleziony w 1945 roku przez **Johna von Neumanna** algorytm **sortowania przez scalanie** jest algorytmem rekurencyjnym. Tak jak powyżej opisaany algorytm – QuickSort - wykorzystuje zasadę **dziel i zwyciężaj**, która polega na podziale zadania głównego na zadania mniejsze dotąd, aż rozwiązanie stanie się oczywiste. Algorytm sortujący dzieli porządkowany zbiór na kolejne połowy dopóki taki podział jest możliwy (tzn. podzbiór zawiera co najmniej dwa elementy). Następnie uzyskane w ten sposób części zbioru rekurencyjnie sortuje tym samym algorytmem. Posortowane części łączy ze sobą za pomocą scalania tak, aby wynikowy zbiór był posortowany.

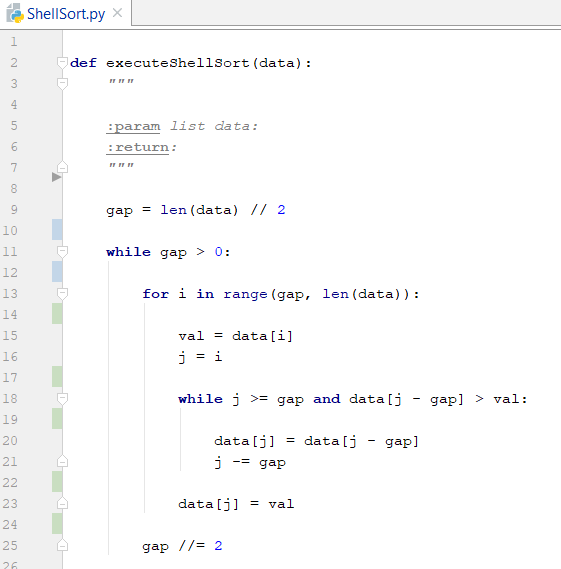
Algorytm sortowania przez scalanie posiada złożoność obliczeniową  dla przypadku typowego oraz optymistycznego jak i pesymistycznego.



***ShellSort***

W latach 50-tych ubiegłego wieku informatyk **Donald Shell**zauważył, iż algorytm [sortowania przez wstawianie](http://eduinf.waw.pl/inf/alg/003_sort/0010.php) (InsertionSort – nie omówiony w tej pracy) pracuje bardzo efektywnie w przypadku gdy zbiór jest w dużym stopniu **uporządkowany**. Z kolei algorytm ten pracuje nieefektywnie w zbiorach **nieuporządkowanych**, ponieważ elementy są przesuwane w każdym obiegu o jedną pozycję przy wstawianiu elementu wybranego na listę uporządkowaną.

Pomysł Shella polegał na tym, iż sortowany zbiór dzielimy na podzbiory, których elementy są odległe od siebie w sortowanym zbiorze o pewien odstęp *h*. Każdy z tych podzbiorów sortujemy algorytmem przez wstawianie. Następnie odstęp zmniejszamy. Powoduje to powstanie nowych podzbiorów (będzie ich już mniej). Sortowanie powtarzamy i znów zmniejszamy odstęp, aż osiągnie on wartość 1. Wtedy sortujemy już normalnie za pomocą Instertion Sort. Jednakże z uwagi na wcześniejsze obiegi sortujące mamy ułatwione zadanie, ponieważ zbiór został w dużym stopniu uporządkowany. Dzięki początkowym dużym odstępom elementy były przesuwane w zbiorze bardziej efektywnie - na duże odległości. W wyniku otrzymujemy najlepszy pod względem szybkości czasu wykonania algorytm sortujący w klasie *O*(*n*2). Algorytm ten nosi również nazwę algorytmu **sortowania przez wstawianie z malejącym odstępem** (ang. Diminishing Increment Sort).



**Porównanie**

Z trzech wybranych sposobów sortowania ShellSort, jak i QuickSort charakteryzują się tym, że sortowanie zachodzi w miejscu – oznacza to, że operacje zostają wykonane na wejściowym ciągu danych przez co nie jest potrzebna większe obciążanie pamięci na pomocnicze podziory „sklejane” w późniejszych operacjach jak w algorytmie sortowania przez sklejanie MergeSort.

MergeSort za to oferuje stabilność egzekucji programu – oznacza to, że w każdym momencie działania programu elementy zbioru posortowane i wstawione na swoje miesca już tam zostają do końca sortowania. Kolejne kroki sortowania nie przesuwają już uporządkowanych elementów i nie „psują” efektów poprzednich operacji.

**Badania**

W testowaniu trzech wybranych algorytmów sortowania, został zmierzony czas bezpośrednio przed rozpoczęciem sortowania, bezpośrednio po jego zakończeniu, a różnica czasu wyrażona w sekundach z dokładnością do szcześciu miejsc po przecinku została zapisana w plikach w ścieszce „Lab3/tests/<algorytm>-<ciąg testowy>-<wielkość ciągu>.txt”

Algorytmy zostały przetestowane 100 razy dla każdego z ciągów wejściowych wyróżniając:

rand - Ciąg o porządku losowym (ang. Random)

sort – Ciąg o porządku rosnącym (ang. Sorted)

half – Ciag w połowie uporządkowaniu rosnąco (ang. Half-Sorted)

rev – Ciąg o porządku malejącym (ang. Reverse-Sorted)

Każdy z tych porządków ciągów występował w wielkościach:

50 tys, 100 tys, 250 tys, 500 tys, 1 mln, 2 mln

Co daje łączną ilość 3x4x6x100 = 7200 testów

**Wyniki**

Wyniki przedstawiają średnią 100 czasów testów każdego z przypadków:

**QuickSort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| QuickSort | | | | |
|  | rand | sort | half | rev |
| 50 | 0.254531 | 0.135156 | 0.340156 | 0.14375 |
| 100 | 0.523125 | 0.282656 | 0.773438 | 0.304063 |
| 250 | 1.45421875 | 0.786563 | 2.141406 | 0.955313 |
| 500 | 3.1528125 | 1.829688 | 4.459531 | 2.062344 |
| 1000 | 6.8775 | 4.237813 | 9.39375 | 4.406875 |
| 2000 | 16.39188 | 10.08922 | 20.99266 | 10.26844 |
|  |  |  |  |  |

**MergeSort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| MergeSort | | | | |
|  | rand | sort | half | rev |
| 50 | 0.465469 | 0.367344 | 0.417344 | 0.373438 |
| 100 | 0.991406 | 0.682188 | 0.775781 | 0.723438 |
| 250 | 2.383125 | 1.854219 | 2.091719 | 1.86 |
| 500 | 4.94484375 | 3.907188 | 4.4375 | 3.917031 |
| 1000 | 10.44859 | 8.256563 | 9.372969 | 8.836406 |
| 2000 | 22.03391 | 17.29781 | 19.75313 | 19.05734 |
|  |  |  |  |  |

**ShellSort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ShellSort | | | | |
|  | rand | sort | half | rev |
| 50 | 0.537188 | 0.169688 | 0.487813 | 0.2825 |
| 100 | 1.365156 | 0.324688 | 1.039375 | 0.5725 |
| 250 | 4.04640625 | 0.897969 | 3.090313 | 1.499375 |
| 500 | 9.609375 | 1.905 | 7.115938 | 3.178438 |
| 1000 | 23.35344 | 4.057969 | 16.26234 | 6.647188 |
| 2000 | 56.3775 | 8.334375 | 37.53516 | 13.73359 |
|  |  |  |  |  |

**Wnioski**

Z przeprowadzonych testów można wyciągnąć wniosek zauważalny praktycznie od razu, tj. Każdy ze sposobów sortowania jest najefektywniejszy i najszybszy dla danych wejściowych już posortowanych do porządku rosnącego, gdzie drugą najszybszą sytuacją jest sortowanie zbioru uporządkowanego malejąco.

Ze wszyskich testów najefektywniejszy okazał się algorytm sortowania szybkiego QuickSort. W przeciwieństwie do dwóch pozostałych, ten sposób najwolniej sortował zbiór uporządkowany w połowie.

QuickSort podczas testów miał jednak jedną wadę, której nie można było pominąć. Właściwy dobór piwotu przed rozpoczęciem sortowania był kluczowy. Dla ciągów nieposortowanych i posortowanych w połowie, piwot należało ustawić na pierwszym miejscu zbioru, a dla ciągów posortowanych (majeląco i rosnąco) należało ustawić go w połowie długości ciągu. Złe ustawienie piwotu (na początku ciągu dla posortowanych zbiorów) skutkowało przepełnieniem stosu, a zmianę egzekucji algorytmu z rekurencujnego na iteracyjny dawało czasy sortowania dla jednego ciągu 50 tys liczb aż 125 sekund, dla 100 tys. Liczb aż 435 sekund, dla 250 tys.ponad 1500 sekund, a dla 500 tys. oraz 1 miliona liczb kolejno 3200s i 7000s co w przełożeniu na godziny daje niemal 2 godziny. Przy przeprowadzeniu wszystkich testów dla samych ciągów uporządkowanych zajęłoby kilka dni. Poprawne ustawienie piwotu dało czasy przedstawione w tabelach oraz na wykresach.

Czas dla sortowania losowo uporządkowanego ciągu przez algorytm Shella był ponad 2 razy dłuższy od pozostałych sposobów sortowania. Jest to cena jaką trzeba zapłacić za fakt, iż jest to algorytm sortujący w miejsciu w przecieństwie do sortowania przez sklejanie MergeSort, co znaczy, że nie jest on pamięciożerne. Z kolei w przeciwieństwie do sortowania szybkiego QuickSort, nie wymaga on „myślenia” przed sortowaniem, aby ustawić piwot w odpowiednim miejsciu.

**Konkluzja**

Każdy z omówionych algorytmów charakteryzuje się dużą efektywnością na tle innych istniejących algorytmów. Każdy z nich może się „pochwalić” wyższością w odpowiednich kategoriach nad pozostałymi dwoma. W używaniu algorytmów sortowania nie należy po prostu włączyć jeden algorytm dla każdego danego zbioru testowego. Każdy z nich (jak i inne, nieomówione w tej pracy) ma swoje wady i superlatywy i powienien być używany z głową, aby dać najefektywniejsze rezultaty dla indywidualnych przypadków.