

Optymalizacja działania algorytmów sortowania - Quick Sort

Mateusz Bencer, nr: 209360

16 kwietnia 2015

Sprawozdanie zawiera wnioski wyciągnięte z optymalizacji algorytmów sortowania - Quick Sort. Celem pomiarów było sprawdzenie wpływu doboru elementu osiowego (ang. pivot) na czas sortowania określonej liczby elementów w kontenerze. Na potrzeby tego zadania użyłem implementacji listy z programu Łukasza Saka. Doświadczenie z pracy nad cudzym kodem okazało się bardzo pouczające...

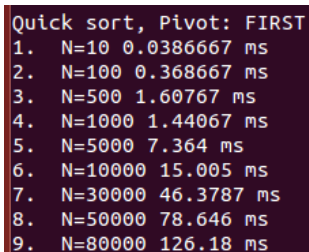
W celu możliwości implementacji algorytmu sortowania konieczne były drobne zmiany w kodzie, tj.:

- Musiałem dodać metody set oraz get do klasy lista, ponieważ niemożliwym było zaimplementowanie algorytmu, gdzie przy pobieraniu elementu został on ściągany, a podczas dodawania inne zostały przesuwane. Metodom pośrednim (sprawdzoną przeze mnie) było pobieranie wszystkich elementów do zwykłej tablicy za pomocą metody pop, posortowanie tej tablicy, i późniejsze powrotne dodanie tych elementów za pomocą metody push do listy. Jednak jak było to do przewidzenia, miało to bardzo negatywny wpływ na złożoność oraz osiągnięte rezultaty czasowe. O wiele prostsza byłaby implementacja tego algorytmu na liście powiązanej (linked list), gdzie każdy element zawiera wskaźnik do poprzedniego i następnego elementu, ale na moment kiedy pobrałem kod z gita, była tam tylko implementacja tablicowa.
- Musiałem dokonać gruntownych zmian w klasie Benchmark, która była stricte dopasowana do poprzedniego zadania (mało uniwersalna).
- Dodałem zaimplementowaną przez siebie wcześniej klasę Timer do pomiarów czasowych.

Zostało przeze mnie przetestowanych 5 sposobów doboru elementu osiowego. Pomiary zostały przeprowadzone dla 9 kolejno rosnących rozmiarach listy. Wszystkie elementy umieszczone w liście są pseudolosowe.

- 1. Pierwszy skrajny element.

Z rozważań teoretycznych wynikało, że jest to najprostsza, ale zarazem najbardziej naiwna metoda doboru elementu osiowego. W przypadku tablicy wstępnie posortowanej taki wybór może skutkować złożonością nawet $O(n^2)$. Moje pomiary (na liście z elementami losowymi) potwierdziły to.

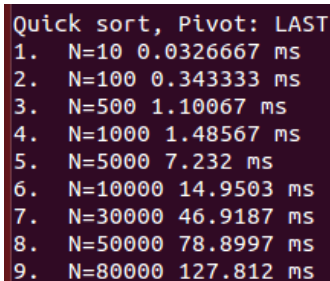


```
Quick sort, Pivot: FIRST
1. N=10 0.0386667 ms
2. N=100 0.368667 ms
3. N=500 1.60767 ms
4. N=1000 1.44067 ms
5. N=5000 7.364 ms
6. N=10000 15.005 ms
7. N=30000 46.3787 ms
8. N=50000 78.646 ms
9. N=80000 126.18 ms
```

Rysunek 1: Wyniki działania programu, gdy piwot jest pierwszym, skrajnym elementem

- 2. Ostatni skrajny element.

Wybór ostatniego elementu jest analogiczny do wyboru pierwszego skaranego elementu. W takim wypadku złożość również może wynieść $O(n^2)$. Moje pomiary również dają podobne rezultaty jak do tych z podpunktu 1.



```
Quick sort, Pivot: LAST
1. N=10 0.0326667 ms
2. N=100 0.343333 ms
3. N=500 1.10067 ms
4. N=1000 1.48567 ms
5. N=5000 7.232 ms
6. N=10000 14.9503 ms
7. N=30000 46.9187 ms
8. N=50000 78.8997 ms
9. N=80000 127.812 ms
```

Rysunek 2: Wyniki działania programu, gdy piwot jest ostatnim, skrajnym elementem

- 3. Element losowy.

Wybór elementu losowego ma na celu uśrednienie możliwości wystąpienia sytuacji najbardziej optymistyczne, jak i najbardziej pesymistycznej. Jest to metoda bardzo zapobiegawcza. Jest złożoność to około $T(n) = 2n \ln(n)$ (przy rozkładzie równomiernym). Jednak według moich pomiarów jest to metoda gorsza od dwóch poprzednich. Podejrzewam, że jest to związane z dodatkowym nakładem czasowym potrzebnym na wylosowanie elementu.

```
Quick sort, Pivot: RANDOM
1. N=10 0.048 ms
2. N=100 0.386667 ms
3. N=500 1.30567 ms
4. N=1000 1.62533 ms
5. N=5000 8.33433 ms
6. N=10000 17.263 ms
7. N=30000 54.5173 ms
8. N=50000 90.7563 ms
9. N=80000 148.93 ms
```

Rysunek 3: Wyniki działania programu, gdy piwot jest elementem losowym

- 4. Element środkowy.

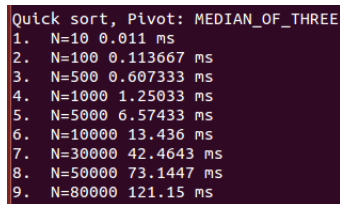
Taki wybór jest chyba najbardziej popularny, przynajmniej wśród implementacji quick sorta znalezionych przeze mnie w internecie. Zapobiega on występowaniu (w prawie wszystkich przypadkach) sytuacji najbardziej pesymistycznej, a wyliczenie elementu środkowego wymaga małych nakładów obliczeniowych. Mimo wszystko uzyskane przeze mnie wyniki są podobne do tych z wyborem elementu pierwszego i ostatniego.

```
Quick sort, Pivot: CENTER
1. N=10 0.0123333 ms
2. N=100 0.133333 ms
3. N=500 0.683333 ms
4. N=1000 1.39167 ms
5. N=5000 7.38 ms
6. N=10000 14.9877 ms
7. N=30000 46.4457 ms
8. N=50000 78.272 ms
9. N=80000 129.781 ms
```

Rysunek 4: Wyniki działania programu, gdy piwot jest elementem środkowym

- 5. Mediana z trzech.

Metoda ta polega na wyliczeniu mediany z elementu najmniejszego, największego i środkowego i jego wyborze, jako elementu osiowego. Idealną sytuacją byłoby za każdym razem obliczanie mediany z całej listy, jednak takie podejście wymaga w praktyce wstępnie posortowanej tablicy do wyznaczenia mediany... (co oczywiście jest sprzeczne z celowością sortowania). Kompromisem dającym dobre rezultaty jest właśnie wyznaczenie mediany z trzech wartości. Według moich pomiarów jest to metoda najszybsza.

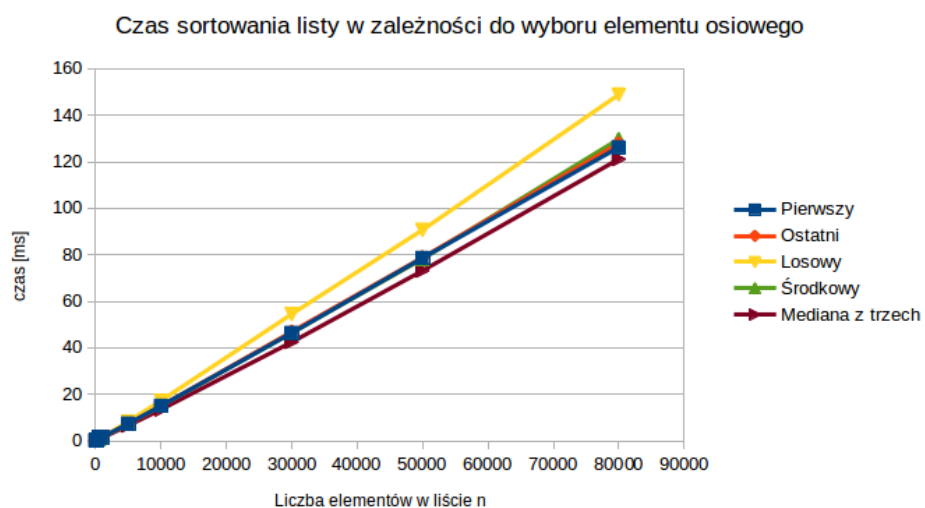


```
Quick sort, Pivot: MEDIAN_OF_THREE
1. N=10 0.011 ms
2. N=100 0.113667 ms
3. N=500 0.607333 ms
4. N=1000 1.25033 ms
5. N=5000 6.57433 ms
6. N=10000 13.436 ms
7. N=30000 42.4643 ms
8. N=50000 73.1447 ms
9. N=80000 121.15 ms
```

Rysunek 5: Wyniki działania programu, gdy piwot jest medianą z trzech elementów

Dla lepszego zobrazowania otrzymanych wyników umieszczam jeszcze wykres ze wszystkimi metodami.

Możemy zaobserwować, że wszystkie metody dają dość podobne rezultaty.



Rysunek 6: Wykres przedstawiający czasy sortowania, przy różnym wyborze piwota

Dla liczb losowych najlepsza jednak (według mojej implementacji) wydaje się być metoda mediany z trzech elementów.