Prowadząca zajęcia:  
dr hab. inż. Małgorzata Sterna, prof. nadzw.

**ALGORYTMY I STRUKTURY DANYCH**

**Ćwiczenie 1**

**Algorytmy Sortowania**

Wojciech Regulski  
Informatyka(WI) I1  
nr 132312

# Cel

Zapoznanie się z zasadą działania różnych algorytmów sortujących. Nauka implementacji poznanych w teorii algorytmów. Uzyskanie wiedzy o praktycznym rozkładzie czasowym wykonywania najbardziej znanych algorytmów sortujących.

# Pomiary

## Zależność czasowa w typowym scenariuszu

Jako typowy scenariusz zostaje uznane sortowanie n elementów z zakresu [1,n] w rozkładzie losowym. (Czas podany w sekundach)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Liczba elementów | Selection Sort | Insertion Sort | Bubble Sort | HeapSort | MergeSort | QuickSort | CountingSort |
| 1000 | 0,001 | 0,001 | 0,003 | Zbyt krótki czas sortowania | | | Zbyt krótki czas sortowania |
| 10000 | 0,144 | 0,146 | 0,241 | 0,002 | 0,002 | 0,001 |
| 20000 | 0,573 | 0,592 | 1,08 | 0,005 | 0,004 | 0,002 |
| 30000 | 1,27 | 1,308 | 2,433 | 0,008 | 0,007 | 0,004 |
| 40000 | 2,241 | 2,328 | 4,355 | 0,012 | 0,01 | 0,006 |
| 50000 | 3,51 | 3,634 | 6,78 | 0,015 | 0,012 | 0,008 |
| 60000 | 5,053 | 5,238 | 9,665 | 0,018 | 0,015 | 0,009 |
| 70000 | 6,866 | 7,076 | 13,203 | 0,022 | 0,017 | 0,011 |
| 80000 | 8,908 | 9,245 | 17,252 | 0,026 | 0,02 | 0,013 |
| 90000 | 11,335 | 11,673 | 22,032 | 0,029 | 0,024 | 0,015 |
| 100000 | 13,948 | 14,387 | 27,356 | 0,033 | 0,028 | 0,016 |
| 1000000 | Zbyt długi czas sortowania | | | 0,211 | 0,16 | 0,101 | 0,025 |
| 2000000 | 0,467 | 0,346 | 0,207 | 0,124 |
| 3000000 | 0,749 | 0,525 | 0,313 | 0,236 |
| 4000000 | 1,041 | 0,727 | 0,421 | 0,352 |
| 5000000 | 1,396 | 0,9 | 0,532 | 0,468 |
| 6000000 | 1,792 | 1,087 | 0,649 | 0,59 |
| 7000000 | 2,145 | 1,263 | 0,752 | 0,71 |
| 8000000 | 2,527 | 1,493 | 0,878 | 0,827 |
| 9000000 | 2,894 | 1,687 | 0,996 | 0,961 |
| 10000000 | 3,297 | 1,905 | 1,121 | 1,081 |

Od razu można zauważyć, że metody sortowania można podzielić na wolne: Selection Sort, Insertion Sort, Bubble Sort; jak i szybkie: Heap Sort, Merge Sort, Quick Sort, Counting Sort. Taki podział wynika ze złożoności obliczeniowej: dla metod wolnych wynosi ona O(n2) zaś dla szybkich O(nlogn) lub O(n).

Wykres ma skalę logarytmiczną, mimo to wyraźnie zarysowuje się ogromna różnica w czasie działania między metodami szybkimi a wolnymi.

Dla większej czytelności dalsza analiza będzie podzielona na dwie części zgodnie z wyżej wymienionym podziałem.

### Metody wolne

|  |  |  |  |
| --- | --- | --- | --- |
| Liczba elementów | SelectionSort | InsertionSort | BubbleSort |
| 10000 | 0,144 | 0,146 | 0,241 |
| 20000 | 0,573 | 0,592 | 1,080 |
| 30000 | 1,270 | 1,308 | 2,433 |
| 40000 | 2,241 | 2,328 | 4,355 |
| 50000 | 3,510 | 3,634 | 6,780 |
| 60000 | 5,053 | 5,238 | 9,665 |
| 70000 | 6,866 | 7,076 | 13,203 |
| 80000 | 8,908 | 9,245 | 17,252 |
| 90000 | 11,335 | 11,673 | 22,032 |
| 100000 | 13,948 | 14,387 | 27,356 |

Na wykresie wyraźnie widać kwadratową zależność czasu od liczby elementów. Uzasadnia to ekstremalną nieefektywność metod wolnych przy dużej ilości elementów. Wykresy Insertion Sort i Selection Sort niemal nakładają się na siebie więc dla losowego rozkładu metody okazują się mieć podobną wydajność, Bubble Sort odstaje nieco od tych dwóch metod.

### Metody szybkie

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Liczba elementów | HeapSort | MergeSort | QuickSort | CountingSort |
| 1000000 | 0,211 | 0,16 | 0,101 | 0,025 |
| 2000000 | 0,467 | 0,346 | 0,207 | 0,124 |
| 3000000 | 0,749 | 0,525 | 0,313 | 0,236 |
| 4000000 | 1,041 | 0,727 | 0,421 | 0,352 |
| 5000000 | 1,396 | 0,9 | 0,532 | 0,468 |
| 6000000 | 1,792 | 1,087 | 0,649 | 0,59 |
| 7000000 | 2,145 | 1,263 | 0,752 | 0,71 |
| 8000000 | 2,527 | 1,493 | 0,878 | 0,827 |
| 9000000 | 2,894 | 1,687 | 0,996 | 0,961 |
| 10000000 | 3,297 | 1,905 | 1,121 | 1,081 |

W metodach szybkich sortowania zależność czasowa jest niemal liniowa. CountingSort traci swoją wielką przewagę w szybkości, gdy zakres sortowanych elementów zaczyna przekraczać milion, przy 10 milionach elementów już prawie zrównując się wydajnościowo z QuickSort. Testy praktyczne potwierdzają tę samą klasę złożoności obliczeniowej dla metod szybkich, mimo iż pomiary wskazują nawet 3-krotną przewagę QuickSort nad HeapSort. Może to wynikać między innymi z implementacji tej metody.

Powyższe pomiary wskazują na efektywność metod szybkich i nieefektywność metod wolnych sortowania. Dla wielkich zbiorów danych sortowanie metodami wolnymi staje się ekstremalnie nieefektywne. Na efektywność metod sortowania ma wpływ ilość elementów, ich rozkład, a także ich zakres – szczególnie ważny przy CountingSort. Metody szybkie korzystają z metody projektowania „dziel i zwyciężaj”, ponieważ bez podzielenia dużego problemu na mniejsze niemożliwe jest efektywne rozwiązanie.

Istnieje też podział na stabilne i niestabilne metody sortowania w zależności czy elementy   
o tej samej wartości przed i po sortowaniu są w takiej samej kolejności względem siebie. Stabilne są: InsertionSort, SelectionSort, BubbleSort, MergeSort, zaś niestabilne są: QuickSort i HeapSort.

## Znaczenie rozkładu liczb na efektywność sortowania

### Rozkład losowy

|  |  |  |  |
| --- | --- | --- | --- |
| Liczba elementów | InsertionSort | QuickSort | QuickSort\_[0] |
| 20000 | 0,243 | 0,002 | 0,002 |
| 40000 | 0,992 | 0,006 | 0,006 |
| 60000 | 2,2 | 0,009 | 0,009 |
| 80000 | 3,906 | 0,013 | 0,013 |
| 100000 | 5,954 | 0,016 | 0,018 |
| 120000 | 8,541 | 0,021 | 0,021 |
| 140000 | 11,901 | 0,024 | 0,024 |
| 160000 | 15,465 | 0,029 | 0,027 |
| 180000 | 19,326 | 0,034 | 0,033 |
| 200000 | 24,18 | 0,036 | 0,036 |

Przy rozkładzie losowym wybranie skrajnego bądź środkowego pivota nie ma większego znaczenia. Raz jeszcze uwidacznia się przepaść w efektywności metod szybkich i wolnych.

### Porządek rosnący

|  |  |  |  |
| --- | --- | --- | --- |
| Liczba elementów | InsertionSort | QuickSort | QuickSort\_[0] |
| 1400 | 0,001 | Zbyt krótki czas sortowania | 0,002 |
| 2800 | 0,004 | 0,008 |
| 4200 | 0,01 | 0,019 |
| 5600 | 0,021 | 0,034 |
| 7000 | 0,03 | 0,052 |
| 8400 | 0,044 | 0,075 |
| 9800 | 0,059 | 0,101 |
| 11200 | 0,074 | 0,134 |
| 12600 | 0,094 | 0,168 |
| 14000 | 0,115 | 0,208 |

Rozłożenie liczb ma ogromny wpływ na efektywność metody QuickSort. Przy rozkładzie losowym pivot może zostać wybrany losowo, jednak przy porządku rosnącym miejsce wybranego pivota ma ogromne znaczenie. Pivot, który ma wartość najmniejszego lub największego elementu jest najgorszym możliwym wyborym sprawiającym, że metoda szybka zaczyna radzić sobie gorzej niż wolna. Dodatkowo może dojść do przepełnienia stosu, co było powodem wybrania tak małego kroku jakim jest 1400 elementów, powyżej 14000 elementów QuickSort wybierający jako pivot element pierwszy przestawał funkcjonować. Dzieje się tak, ponieważ przy pivocie będącym najmniejszym elementem tablica nie zostaje podzielona na dwie tablice średniej wielkości tylko tablicę jednoelementową i całą resztę. QuickSort biorący za pivot element środkowy radzi sobie za to bardzo dobrze – jest to dla niego najlepsze ustawienie elementów. Podział następuje w połowie, duży problem zostaje podzielony na dwa równe średnie problemy. Dla InsertionSort porządek rosnący również jest najlepszym przypadkiem, w którym znacząco przyspiesza, ponieważ potrzebuje zaledwie sprawdzić czy każdy element jest na swoim miejscu.

Istnieją również inne metody wybraniu pivota. Jedną z najlepszych jest wybranie losowego elementu za pivot. Pozwala to ograniczyć do minimum możliwość trafienia na najgorszy scenariusz danych wejściowych, niestety spada wydajność przy wstępnie posortowanych danych względem pivota środkowego. Inną metodą jest wybranie przybliżonej mediany za pivot. W tym przypadku sortowanie staje się bardzo szybkie i ma mało zagnieżdżony stos, jednak wybranie pivota wiąże się ze sporymi obliczeniami, co w rezultacie prowadzi do spadku wydajności całego procesu. Zagnieżdżenie stosu jest ważnym aspektem, ponieważ w skrajnych przypadkach może prowadzić do przepełnienia stosu, zaś w mniej skrajnych do znacznego zwiększenia złożoności pamięciowej, ponieważ QuickSort wymaga dodatkowej pamięci właśnie na stos.

Wybór między QuickSort a InsertionSort zależy od przewidzianych danych wejściowych na jakich będą pracować. Należy mieć na uwadzę duża zależność efektywności QuickSorta od rozłożenia danych wejściowych w różnych sposobach wybrania pivota. W miejscach gdzie wymagana jest stabilność szybkości działania QuickSort nie będzie dobrym wyborem przez jego problemy z sortowaniem w pesymistycznym przypadku. W ogólnym rozrachunku QuickSort mimo kilku problemów jest niezwykle szybką metodą sortowania. InsertionSort bardzo dobrze radzi sobie z danymi wejściowymi, które są już wstępnie posortowane, jest to jego główna zaleta.   
W innych przypadkach nie jest już zbyt efektywny należąc do metod wolnych.

InsertionSort:   
zalety: duża efektywność przy wstępnie posortowanych danych; stabilny  
wady: kwadratowa złożoność obliczeniowa

SelectionSort:  
zalety: mało zamian elementów (przydatne za nośnikach z bardzo niską prędkością zapisu); stabilny  
wady: kwadratowa złożoność obliczeniowa

BubbleSort:  
zalety: w implementacji z flagą wysoka efektywność przy wstępnie posortowanych danych; stabilny  
wady: w typowym przypadku gorsza wydajność nawet od innych metod wolnych

QuickSort:  
zalety: najszybsza metoda w typowym scenariuszu  
wady: bardzo mała wydajność w pesymistycznym scenariuszu; problem wybrania pivota; potrzebna dodatkowa pamięć na stos; niestabilny

MergeSort:  
zalety: metoda niewiele wolniejsza od QuickSort; pesymistyczna złożoność nlogn; stabilny  
wady: dodatkowa tablica wynikowa – n dodatkowej pamięci

HeapSort:  
zalety: pesymistyczna złożoność nlogn; algorytm w miejscu  
wady: niestabilny

CountingSort:  
zalety: bezkonkurencyjny przy małym zakresie liczb  
wady: wąskie zastosowanie; wysoka złożoność pamięciowa

## Znaczenie zakresu liczb na efektywność sortowania

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Liczba elementów | 100n | | 0,01n | |
| QuickSort | CountingSort | QuickSort | CountingSort |
| 200000 | 0,017 | 0,068 | 0,013 | 0,001 |
| 400000 | 0,036 | 0,137 | 0,024 | 0,001 |
| 600000 | 0,056 | 0,212 | 0,039 | 0,002 |
| 800000 | 0,075 | 0,29 | 0,056 | 0,003 |
| 1000000 | 0,096 | 0,354 | 0,069 | 0,004 |
| 1200000 | 0,114 | 0,454 | 0,087 | 0,006 |
| 1400000 | 0,137 | 0,533 | 0,098 | 0,008 |
| 1600000 | 0,159 | 0,639 | 0,112 | 0,009 |
| 1800000 | 0,181 | 0,682 | 0,13 | 0,01 |
| 2000000 | 0,207 | 0,784 | 0,148 | 0,012 |
| 2200000 | 0,221 | 0,861 | 0,164 | 0,013 |
| 2400000 | 0,253 | 0,934 | 0,185 | 0,015 |
| 2600000 | 0,272 | 1,033 | 0,2 | 0,016 |
| 2800000 | 0,298 | 1,099 | 0,215 | 0,019 |
| 3000000 | 0,316 | 1,2 | 0,225 | 0,022 |

### Zakres 100 razy większy niż ilość elementów

### Zakres 100 razy mniejszy niż ilość elementów

Doświadczenie pokazuje jak wielkie znaczenie ma zakres elementów dla metody CountingSort. CountingSort nie radzi sobie zbyt dobrze, gdy zakres zaczyna być duży, przegrywa nawet z QuickSort, który ma gorszą złożoność obliczeniową. Co również ważne CountingSort uzyskuje wtedy ogromną złożoność pamięciową. Po zmniejszeniu zakresu radzi on już sobie bardzo dobrze, będąc zdecydowanie szybszym niż QuickSort. QuickSort również zyskał na zmniejszeniu zakresu liczb, jednak w niewielkim stopniu.