Badanie efektywności wybranych algorytmów sortowania ze względu na złożoność obliczeniową

AiZO - zadanie projektowe nr 1

Autor : Arkadiusz Błasiak

Nr albumu: 273022

Data: 18.04.2024 godz. 11.15

# Wprowadzenie

### Pojęcie złożoności obliczeniowej

Złożoność obliczeniowa algorytmu określa, jak wydajny jest algorytm, ile musi on wykonać operacji w zależności ilości danych oraz ile potrzebuje do tego pamięci. Często zdarza się, że dany problem algorytmiczny można rozwiązać kilkoma metodami, czyli algorytmami o różnej złożoności obliczeniowej. Złożoność obliczeniową dzielimy na złożoność pamięciową oraz złożoność czasową.

W tym projekcie pomijamy złożoność pamięciową i skupiamy się w pełnie na złożoności czasowej.

### Kilka ważnych pojęć:

* złożoność czasowa

ilość czasu potrzebnego do wykonania zadania, wyrażona jako funkcja ilości danych.

* złożoność obliczeniowa

ilość zasobów komputerowych potrzebnych do wykonania zadania.

* złożoność oczekiwana

inaczej złożoność średnia; ilość zasobów potrzebna do zrealizowania zadania dla statystycznie oczekiwanych danych; zapisywana za pomocą notacji theta –

* złożoność pesymistyczna

ilość zasobów potrzebna do zrealizowania zadania w przypadku najgorszych danych; zapisywana za pomocą notacji „duże O” –

# Algorytmy sortowania

## Sortowanie przez wstawanie (Insertionsort)

Jeden z najprostszych algorytmów sortowania, którego zasada działania odzwierciedla sposób w jaki ludzie ustawiają karty – kolejne elementy wejściowe są ustawiane na odpowiednie miejsca docelowe. Jego zalety: jest stabilny, jest wydajny dla zbiorów o niewielkiej liczebności, liczba wykonanych porównań jest zależna od liczby inwersji w permutacji, dlatego algorytm jest wydajny dla danych wstępnie posortowanych.

Złożoność oczekiwana:

Złożoność pesymistyczna:

## Sortowanie Shella (Shellsort)

Jeden z algorytmów sortowania działających w miejscu i korzystających z porównań elementów. Można go traktować jako uogólnienie sortowania przez wstawianie lub sortowania bąbelkowego, dopuszczające porównania i zamiany elementów położonych daleko od siebie. Na początku sortuje on elementy tablicy położone daleko od siebie, a następnie stopniowo zmniejsza odstępy między sortowanymi elementami. Dzięki temu może je przenieść w docelowe położenie szybciej niż zwykłe sortowanie przez wstawianie.

Złożoność wersji Shella:

Złożoność oczekiwana:

Złożoność pesymistyczna:

Złożoność wersji Knutha:

Złożoność oczekiwana: brak danych

Złożoność pesymistyczna:

## Sortowanie przez kopcowanie (Heapsort)

Jeden z algorytmów sortowania, choć niestabilny, to jednak szybki i niepochłaniający wiele pamięci. Podstawą algorytmu jest użycie kolejki priorytetowej zaimplementowanej w postaci binarnego kopca zupełnego. Zasadniczą zaletą kopców jest stały czas dostępu do elementu maksymalnego (lub minimalnego) oraz logarytmiczny czas wstawiania i usuwania elementów; ponadto łatwo można go implementować w postaci tablicy.

Złożoność oczekiwana:

Złożoność pesymistyczna:

## Sortowanie szybkie (Quicksort)

jeden z popularnych algorytmów sortowania działających na zasadzie „dziel i zwyciężaj”. Algorytm sortowania szybkiego jest wydajny. Ze względu na szybkość i prostotę implementacji jest powszechnie używany. Jego implementacje znajdują się w bibliotekach standardowych wielu środowisk programowania.

Złożoność oczekiwana:

Złożoność pesymistyczna:

# Metody

1. Program zapisany został w języku C++ wykorzystując programowanie obiektowe,
2. Tablice do posortowania tworzone są dynamicznie,
3. Tablice są zapełnione liczbami typu int. Wyjątek stanowi sortownie przez kopcowanie, gdzie wykorzystuje się również tablice z typem float.
4. Program uruchamiany był na laptopie HP Pavilion – 15-bc402nw z procesorem Intel Core i5-8300H, kartą graficzną Nvidia GeForece GTX 1050 oraz 16 GB pamięci RAM,
5. Czas mierzony był, kiedy laptop podłączony był do prądu, w trybie pełnej wydajności, minimalizując udział niepotrzebnych procesów w tle,
6. Do odmierzania czasu program posługuje się funkcją std::chrono::high\_resolution\_clock z biblioteki chrono,
7. Czas mierzony był mikrosekundach i zaokrąglany do pełnych milisekund za wyjątkiem sytuacji, kiedy pierwsza znacząca cyfra była w ułamku
8. Docelowo program średni czas mierzył wykonując 50 sortowań wyciągając z tego średnią arytmetyczną. Jest to kompromis wyciągnięty z wykresu ilustrujący, iż od już poniżej tej wartości wyniku zaczynają się unormowywać:

Jednakże, kiedy czasy rosły różnica pomiędzy poszczególnymi wynikami mała poniżej 0,5%, jak w poniższym przypadku, kiedy zbadano jak zmienia się średnia dodając kolejne wyniki do wzoru, dla 32 elementowej tablicy posortowanej w 66% sortując algorytmem przez wstawianie:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| średnia | 6232 | 6236 | 6262 | 6243 | 6226 | 6217 | 6209 | 6205 | 6200 | 6197 |

Kiedy czasy są na poziomie dziesiątek czy też nawet setek sekund, fluktuacje na poziomie sekund i milisekund są pomijalne. Dlatego też brana była średnia z 5, 3, a nawet 1 wyniku, kiedy to wynik czas potrzebny na odczytanie wyniku był długi.

# Pomiary

## Sortowanie przez wstawanie (Insertionsort)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| elements [10^3] | time [ms] | | | | |
| random | sorted 33% | sorted 66% | sorted | sorted desc |
| 10 | 55 | 24 | 6 | 0,03 | 108 |
| 20 | 218 | 97 | 26 | 0,06 | 422 |
| 40 | 856 | 378 | 102 | 0,12 | 1677 |
| 80 | 3415 | 1500 | 398 | 0,25 | 6716 |
| 160 | 13981 | 5986 | 1578 | 0,48 | 26699 |
| 320 | 54528 | 24011 | 6197 | 0,96 | 106961 |
| 640 | 219694 | 95890 | 24733 | 1,88 | 432132 |
| 1 280 | 858402 | 383674 | 98744 | 3,73 | 1716160 |

## Sortowanie przez scalanie (Shellsort)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | time [ms] | | | | | | | | | |
| elements [10^3] | Shell version | | | | | Knuth version | | | | |
| random | sorted 33% | sorted 66% | sorted | sorted desc | random | sorted 33% | sorted 66% | sorted | sorted desc |
| 10 | 1 | 1 | 0,8 | 0,5 | 0,7 | 1 | 1 | 0,6 | 0,3 | 0,5 |
| 20 | 3 | 2 | 1 | 1 | 1 | 3 | 2 | 1 | 0,6 | 1 |
| 40 | 7 | 5 | 3 | 2 | 3 | 6 | 5 | 3 | 1 | 2 |
| 80 | 16 | 12 | 8 | 5 | 6 | 14 | 11 | 7 | 3 | 5 |
| 160 | 35 | 26 | 17 | 11 | 14 | 33 | 24 | 15 | 7 | 10 |
| 320 | 80 | 56 | 38 | 23 | 29 | 74 | 51 | 32 | 14 | 20 |
| 640 | 175 | 124 | 82 | 49 | 62 | 158 | 112 | 68 | 29 | 42 |
| 1 280 | 396 | 285 | 184 | 106 | 131 | 351 | 252 | 148 | 65 | 88 |

## Sortowanie przez kopcowanie (Heapsort)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | time [ms] | | | | | | | | | |
| elements [10^3] | integer | | | | | float | | | | |
| random | sorted 33% | sorted 66% | sorted | sorted desc | random | sorted 33% | sorted 66% | sorted | sorted desc |
| 10 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 2 | 2 | 2 |
| 20 | 5 | 5 | 5 | 5 | 4 | 5 | 6 | 5 | 5 | 5 |
| 40 | 11 | 11 | 11 | 10 | 9 | 12 | 12 | 11 | 10 | 10 |
| 80 | 25 | 25 | 23 | 21 | 20 | 26 | 26 | 24 | 21 | 20 |
| 160 | 53 | 53 | 50 | 44 | 41 | 55 | 55 | 51 | 44 | 42 |
| 320 | 115 | 111 | 103 | 92 | 86 | 119 | 117 | 106 | 91 | 88 |
| 640 | 243 | 236 | 215 | 186 | 179 | 250 | 249 | 225 | 189 | 183 |
| 1 280 | 515 | 502 | 448 | 373 | 375 | 531 | 534 | 476 | 396 | 385 |

## Sortowanie szybkie (Quicksort)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | time [ms] | | | | | | | | | |
| elements [\*10^3] | left pivot | | | | | right pivot | | | | |
| random | sorted 33% | sorted 66% | sorted | sorted desc | random | sorted 33% | sorted 66% | sorted | sorted desc |
| 10 | 1 | 50 | 80 | 90 | 83 | 1 | 9 | 35 | 78 | 73 |
| 20 | 3 | 190 | 303 | 340 | 298 | 3 | 32 | 123 | 277 | 242 |
| 40 | 5 | 693 | 1119 | 1270 | 990 | 5 | 110 | 421 | 952 | 747 |
| 80 | 11 | 2341 | 3849 | 4240 | 3047 | 11 | 372 | 1461 | 3329 | 2277 |
| 160 | 24 | 7037 | 11362 | 13064 | 8371 | 24 | 1233 | 4878 | 11110 | 6969 |
| 320 | 49 | 19255 | 30916 | 35070 | 21650 | 48 | 3629 | 14429 | 33032 | 19496 |
| 640 | 100 | 48755 | 78521 | 88528 | 53167 | 99 | 9579 | 37498 | 86254 | 49909 |
| 1 280 | 201 | 119242 | 192757 | 219592 | 125890 | 200 | 23215 | 93420 | 214934 | 121115 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | time [ms] | | | | | | | | | |
| elements [\*10^3] | middle pivot | | | | | random pivot | | | | |
| random | sorted 33% | sorted 66% | sorted | sorted desc | random | sorted 33% | sorted 66% | sorted | sorted desc |
| 10 | 0,9 | 1 | 0,7 | 0,4 | 0,4 | 2 | 2 | 1 | 1 | 1 |
| 20 | 3 | 2 | 1 | 0,9 | 0,9 | 4 | 3 | 3 | 2 | 2 |
| 40 | 5 | 4 | 3 | 2 | 2 | 8 | 7 | 6 | 6 | 5 |
| 80 | 11 | 9 | 7 | 5 | 5 | 16 | 14 | 14 | 12 | 12 |
| 160 | 24 | 19 | 14 | 10 | 10 | 33 | 29 | 27 | 25 | 25 |
| 320 | 48 | 38 | 28 | 19 | 20 | 67 | 60 | 55 | 49 | 51 |
| 640 | 98 | 77 | 58 | 41 | 43 | 136 | 124 | 113 | 103 | 103 |
| 1 280 | 197 | 157 | 121 | 87 | 92 | 272 | 268 | 245 | 229 | 229 |

# Podsumowanie i wnioski

Dla wyników sortowania przez wstawianie można zobaczyć wyraźną funkcję kwadratową. Jest on wyraźnie najwolniejszy ze wszystkich mierzonych, dlatego jego implementacja nie jest sugerowana w literaturze. Z plusów jednak bardzo dobrze radzi sobie z tablicami wstępnie i całkowicie posortowanymi, ponieważ nie wykonuje wtedy żadnych zamian, dlatego też jest on wtedy nieporównywalnie szybszy od elementów czysto losowych, co można wyraźnie dostrzec na wykresach – im tablica zawiera więcej elementów posortowanych, tym algorytm zaczyna działać znacznie szybciej.

Algorytm sortowania przez scalanie jest znacznie szybszy od poprzednika, czasy nie rosną w wysokim tempie, co może wykazywać zależność quasi-logarytmiczną. Algorytm Knutha jest zgodnie z przewidywaniami nieco szybszy od wersji Shella. Dla obydwu przypadków na korzyść gra posortowanie tablic, które również potrafią się dość znacząco przyczynić do zmniejszenia czasów działania algorytmów.

# Źródła

* <https://www.algorytm.edu.pl/matura-informatyka/zlozonosc-algorytmu>
* <https://zpe.gov.pl/a/przeczytaj/DNDIkCVVg>
* <https://wikipedia.org/>