Autor:

DAWID PICHEN (132775)

PROJEKT Z TEORETYCZNYCH PODSTAW INFORMATYKI

Temat:

PORÓWNANIE ALGORYTMÓW SORTOWANIA

Wstęp

Celem niniejszego projektu było porównanie i wskazanie złożoności obliczeniowych czterech wybranych algorytmów sortowania. Wybrałem następujące algorytmy:

- algorytm sortowania bąbelkowego,
- algorytm sortowania przez wstawianie,
- algorytm sortowania kopcowego (stogowego),
- algorytm sortowania szybkiego QuickSort.

Opis programu

Porównanie powyższych algorytmów polegało na mierzeniu czasu, jaki jest potrzebny na posortowanie danych w tablicy w kolejności rosnącej dla każdego z algorytmów. Badałem czas sortowania dla następujących wielkości tablic: 10, 50, 100, 500, 1000, 2500, 5000, 7500, 10000, 15000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000 i 100000 elementów. Każdy element w tablicy był liczbą całkowitą bez znaku (*unsigned int*), liczba taka jest reprezentowana w pamięci w postaci 4 bajtów (kompilator GCC dla procesora i systemu 32-bitowego).

Postanowiłem napisać program testujący algorytmy sortowania w języku ANSI C, gdyż zależało mi na możliwie szybkiej pracy programu. Aplikacja została napisana dla systemu operacyjnego Linux, jednakże nie powinno być większych problemów z jej skompilowaniem w innych systemach UNIXowych. Program składa się z 4 plików z kodami źródłowymi i 3 plików nagłówkowych. Plik *main.c* zawiera główną pętlę programu odpowiedzialną za uruchamianie operacji sortowania dla różnych algorytmów oraz dla różnych rozmiarów tablic. Dla każdego rozmiaru tablic aplikacja wykonuje *L_PROB* prób sortowania (domyślnie 10, definiowane w pliku *czas.h*). Dla każdej próby program alokuje dwie tablice liczb. Pierwsza tablica wypełniania jest liczbami losowymi, druga natomiast służy jako kopia pierwszej tablicy. Kopia tablicy jest niezbędna, gdyż porównywanie czasów sortowania powinno być realizowane na tych samych tablicach danych. Po każdym sortowaniu pierwsza tablica (*tab1*) zawiera posortowane dane, dlatego przed następnym sortowaniem innym algorytmem, trzeba przywrócić oryginalne dane zawarte w tablicy *kopia*. Plik *sort.c* zawiera funkcje sortujące dla badanych algorytmów sortowania. W pliku *tab.h* znajduje się definicja struktury *Tablica*. Struktura ta przechowuje informację o liczbie elementów w tablicy, jak również wskaźnik na pierwszy element tej tablicy. Plik *tab.c* zawiera funkcje związane z obsługą tablic, znajduje się tu m.in. funkcja wypełniająca tablicę

liczbami losowymi. Liczby te są generowanea za pomocą funkcji *rand()* z biblioteki standardowej. Dla biblioteki GNU wartości zwracane przez tę funkcję należą do przedziału <0, 2147483647>. Z kolei plik *czas.c* zawiera funkcje związane z pomiarem czasu, jak również obliczaniem wartości średniej z *L_PROB* prób sortowań.
Zależało mi na jak najdokładniejszym pomiarze czasu. Nie mogłem użyć funkcji *time()*, gdyż funkcja ta zwracała czas tylko z dokładnością do sekundy. Postanowiłem wykorzystać w tym celu systemową funkcję *gettimeofday(struct timeval *tp, void *tzp)*. Jej pierwszym argumentem jest wskaźnik na strukturę *timeval* opisaną w pliku *sys/time.h*, struktura ta zawiera dwa składniki:
long tv_sec (liczba sekund, które upłynęły od początku epoki UNIXowej, czyli 01.01.1970)
long tv_usec (mikrosekundy).
Drugi argument to wskaźnik na strukturę opisującą strefę czasową. Dla pomiarów czasu jest on nieistotny (*NULL*). Widać, że funkcja systemowa pozwala na pomiar czasu z dokładnością do mikrosekund.
Program wyświetla wyniki czasów sortowań na ekranie w sposób szczegółowy, tzn. dla każdej próby. Tworzy on także plik *wyniki.txt* w katalogu w którym został uruchomiony. Plik ten zawiera wyniki średnich czasów sortowań dla każdego zadanego rozmiaru tablicy.

Kompilacja programu

Dołączony do projektu dysk zawiera pliki źródłowe, jak również skompilowany plik wykonywalny programu. Uruchamiamy program poleceniem *./projekt* . Jeśli jednak chcemy skompilować program ze źródeł, to najpierw należy go wypakować z archiwum poleceniem:
tar -zxvf projekt.tar.gz
następnie uruchamiamy kompilację poleceniem *make*.

Analiza złożoności obliczeniowych

W kodzie algorytmu sortowania bąbelkowego występują dwie pętle programu, w taki sposób, że w głównej pętli realizowana jest druga pętla. Pierwsza pętla zgodnie z kodem zawartym w pliku *sort.c* wykonuje się n-1 razy, natomiast wewnętrzna pętla wykonuje się n-2 razy. A zatem złożoność pierwszej pętli jak i pętli drugiej w przybliżeniu wynosi n, zatem ogólna złożoność obliczeniowa tego algorytmu to $O(n^2)$.

Kod algorytmu sortowania przez wstawianie zawiera również dwie pętle, w której jedna wykonuje się we wnętrzu drugiej. Pierwsza pętla wykonuje się n-1 razy, natomiast pętla wewnętrzna wykonuje się od n-2 razy do jednego razu. Ogólna złożoność obliczeniowa tego algorytmu to $O(n \cdot n) = O(n^2)$.

Złożoność obliczeniowa algorytmu sortowania kopcowego. Na początku kopiec jest roboczo kopcowany. Widać, że operacja ta wykonuje się w pętli ½(n-1) razy (bo w kodzie *i = n>>1*, to przesunięcie bitowe w prawo o jedno miejsce, czyli dzielenie bez reszty). A zatem złożoność tego kroku wynosi *n*. W pętli tej wykonuje się dodatkowa pętla, której złożoność teraz przeanalizujemy. Zbadajmy kopiec w postaci graficznej. Niech *k* oznacza liczbę poziomów kopca, a *n* niech oznacza liczbę elementów. Oczywistym jest, że:
$2^0+2^1+2^2+2^3+\ldots+2^{k-1} = n$
Korzystając ze wzoru na sumę szeregu otrzymujemy: $2^k - 1 = n$, czyli $k = log_2(n+1)$.
A zatem łącząc te dwie złożoności obliczeniowej, otrzymujemy, że ogólna złożoność obliczeniowa algorytmu sortowania kopcowego wynosi $O(n \cdot log(n+1)) = O(n \cdot log n)$.

Algorytm QuickSort został wymyślony przez sira Charles'a Antony'ego Richarda Hoare'a w roku 1960. Jego złożoność obliczeniowa w najgorszym wypadku jest równa $O(n^2)$, jednakże w praktyce jego złożoność obliczeniowa wynosi $O(n \cdot log n)$. Co ciekawe, praktycznie jest on szybszy od pozostałych algorytmów które mają złożoność $O(n \cdot log n)$.

Oczywiście złożoność obliczeniowa O(n·logn) jest znacznie lepsza od złożoności $O(n^2)$.

Badanie czasów sortowań

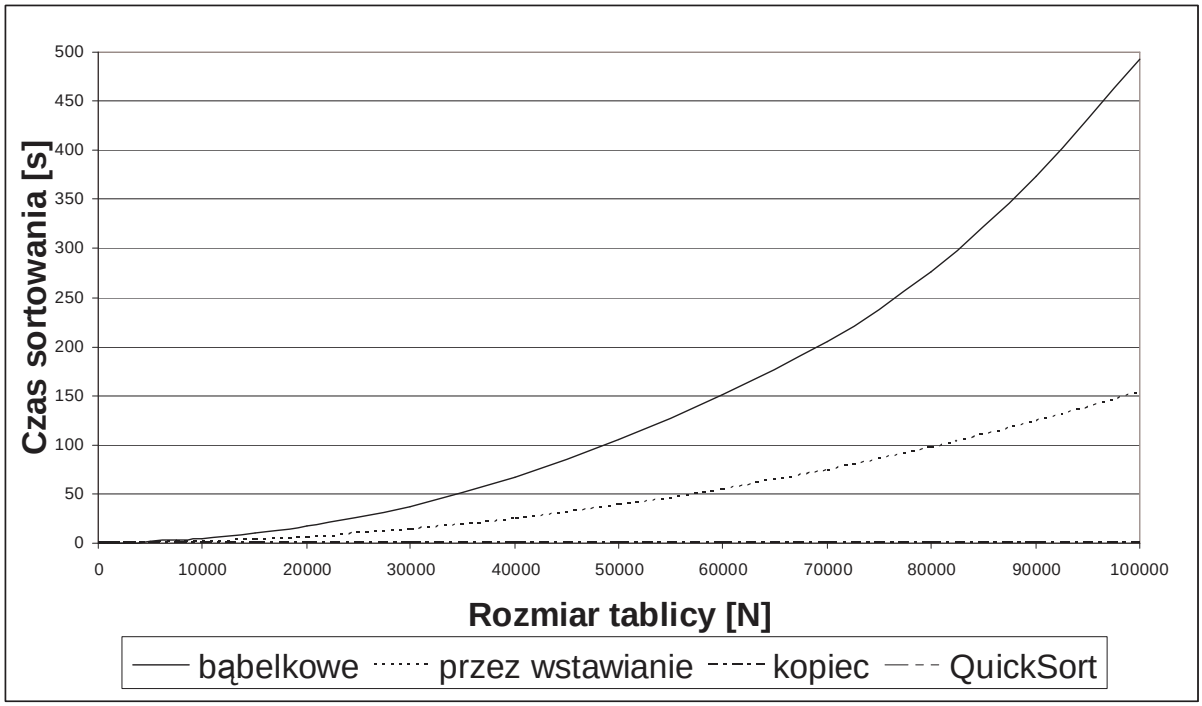
Uruchomiłem program mierzący czasy algorytmów sortowania na następującym systemie komputerowym:

- procesor: Intel Pentium III 450 MHz,
- pamięć RAM: 160 MB SDRAM PC-100,
- system operacyjny: Slackware Linux 10,
- jądro systemu: 2.4.24.

Przed uruchomieniem programu zamknąłem wszystkie zbędne procesy, które potencjalnie mogłabym spowalniać pracę aplikacji, pozostawiłem tylko niezbędne procesy systemowe. Program został uruchomiony pod zwykłą konsolą tekstową (serwer X wyłączony). Uruchomiłem program na noc, następnego dnia uzyskałem następujące wyniki (załączone na dyskietce).

Rozmiar tablicy [N]	**Średnie czasy sortowań [s] dla algorytmów sortowania:**			
	bąbelkowego	**przez wstawianie**	**kopcowego**	**QuickSort**
10	0,000006	0,000004	0,000004	0,000005
50	0,000099	0,000043	0,000025	0,000025
100	0,000383	0,000156	0,000056	0,000057
500	0,009638	0,003747	0,000372	0,000347
1000	0,038313	0,014808	0,000830	0,000769
2500	0,237749	0,091923	0,002350	0,002199
5000	1,041387	0,369902	0,005172	0,004877
7500	2,362887	0,840565	0,008316	0,007550
10000	4,199384	1,502687	0,011622	0,010427
15000	9,440003	3,388880	0,018616	0,016722
20000	16,782126	6,044942	0,026045	0,022371
30000	37,733099	13,613930	0,041641	0,035871
40000	67,108282	24,275645	0,057960	0,049042
50000	104,832556	37,905661	0,075031	0,062260
60000	151,021899	54,681647	0,092323	0,075445
70000	205,462451	74,271826	0,110057	0,088930
80000	275,654443	97,203198	0,128523	0,104802
90000	373,857007	123,530872	0,147742	0,119675
100000	493,331982	153,549268	0,168172	0,131421

Otrzymane wyniki w tabeli:



Wnioski:

1. Dla względnie małych rozmiarów tablic, tj. do 50 elementów czasy sortowań są prawie takie same, tzn. są tego samego rzędu i wynoszą dziesiątki mikrosekund.
2. Dla coraz większych rozmiarów tablic sortowanie bąbelkowe pochłania drastycznie coraz większych czasów. Np. dla tablicy 50000 elementowej, czas wymagany do posortowania wynosi ok. 100 sekund, podczas gdy dla tablicy o 100% większej (tj. 100000 elementowej) czas sortowania wynosi niecałe 500 sekund (czyli jest o 400% większy niż czas potrzebny na posortowanie o połowę mniejszej tablicy).
3. Sortowanie przez wstawianie wygląda korzystniej niż sortowanie bąbelkowe, jednakże wciąż trwa długo. Jak widać z wykresu, czas sortowania rośnie mniej więcej liniowo ze wzrostem rozmiaru tablicy. Sortowanie kopcowe prezentuje się znaczniej lepiej od poprzednich algorytmów. Mimo, że tablica zawierała bardzo dużo liczbę elementów (np. 100000), to czas sortowania był mniejszy niż 1 s., podczas gdy sortowanie bąbelkowe takiej samej tablicy potrzebowało niecałe 500 sekund.
4. Bezsprzecznie najlepszym algorytmem sortowania okazał się algorytm QuickSort, który tablicę 100000 elementową posortował w czasie zaledwie 13 ms. Wynik ten był jednak niewiele lepszy od sortowania kopcowego (sortowanie tej samej tablicy zajęło 17 ms).

Projekt ten wykazał, że pisząc jakiekolwiek programy nie powinno się stosować prostych algorytmów, takich jak sortowanie bąbelkowe, czy też sortowanie przez wstawianie. Znaczniej lepiej posłużyć algorytmem QuickSort, gdyż jest on bardzo szybki i nie jest trudny w implementacji.