POLITECHNIKA KRAKOWSKA		
WYDZIAŁ INŻYNIERII ELEKTRYCZNEJ I KOMPUTEROWEJ		
METODY PROGRAMOWANIA – PROJEKT		
Data oddania: 13.06.2024 r.	**Temat projektu:** Algorytmy sortowania	**Wykonał zespół w składzie:** Dominik Koralik Łukasz Konieczny Maciej Jankoś

1. WSTĘP I CEL

Cel: Naszym zadaniem było napisanie programu, którego zadaniem było sortowanie tablicy liczb poprzez 4 różne sortowania. Użytkownik w menu miał mieć możliwość podania tablicy liczb w pliku tekstowym, oraz wybrania jednego z 4 możliwych sortowań: MergeSort, QuickSort, BubbleSort oraz HeapSort. Po wybraniu jednej z możliwości, program miał się kompilować, wyniki przesyłać do osobnego pliku tekstowego, oraz miał mierzyć tzw. processor time oraz work time.

Wstęp teoretyczny

Algorytmy sortowania - Problem sortowania można zdefiniować następująco:

- Danymi wejściowymi jest ciąg *n* liczb.
- Wynikiem jest ciąg tych liczb w kolejności posortowanej rosnąco lub malejąco.

Zadaniem algorytmu sortowania jest takie przestawienie elementów danego ciągu, aby były one uporządkowane rosnąco lub malejąco. W naszym programie użyliśmy 4 algorytmów sortowania, tj.: przez scalanie, przez kopcowanie, przez sortowanie bąbelkowe oraz przez sortowanie szybkie. Po krótce opiszemy każdy z nich.

a) MergeSort (sortowanie przez scalanie)

Sortowanie przez scalanie należy do grupy algorytmów sortowania szybkich. Na koniec algorytm ustawia elementy w tablicy pierwotnej w porządku leksykograficznym. Zasad działania algorytmu polega na podziale tablicy na elementy 1- wyrazowe, które uważane są już za posortowane. Następną fazą jest porównywanie elementów oraz scalenie tablicy.

PSEUDOKOD

```
1. funkcja mergeSort(tablica, lewy, prawy):
2. jeśli lewy < prawy:
3. środek = (lewy + prawy) / 2
4. mergeSort(tablica, lewy, środek) // Sortuj lewą podtablicę rekurencyjnie
5. mergeSort(tablica, środek + 1, prawy) // Sortuj prawą podtablicę rekurencyjnie
6. scal(tablica, lewy, środek, prawy) // Scal posortowane podtablice
```

b) BubbleSort (sortowanie bąbelkowe)

Bubblesort to prosty algorytm sortowania, który polega na wielokrotnym przechodzeniu przez listę elementów, porównywaniu sąsiednich elementów i zamienianiu ich miejscami, jeśli są w niewłaściwej kolejności. Proces ten powtarza się aż do momentu, gdy lista jest w pełni posortowana.

PSEUDOKOD

```
funkcja bubblesort(tablica)
n = długość(tablicy)
for i od 0 do n-1 wykonaj
for j od 0 do n-2-i wykonaj
if tablica[j] > tablica[j+1] wtedy
zamień(tablica[j], tablica[j+1])
koniec if
koniec for
koniec for
koniec funkcji

funkcja zamień(a, b)
temp = a
a = b
b = temp
koniec funkcji
```

c) QuickSort (sortowanie szybkie)

Quicksort, czyli szybkie sortowanie, to efektywny algorytm sortowania. Polega na dzieleniu listy na mniejsze podlisty na podstawie wybranego elementu zwanego

pivotem (osią), a następnie sortowaniu tych podlist. Działa na zasadzie „Dziel i zwyciężaj”.

PSEUDOKOD

```
funkcja quicksort(tablica, lewy, prawy)
jeśli lewy < prawy wtedy
pivotIndex = podziel(tablica, lewy, prawy)
quicksort(tablica, lewy, pivotIndex - 1)
quicksort(tablica, pivotIndex + 1, prawy)
koniec jeśli
koniec funkcji

funkcja podziel(tablica, lewy, prawy)
pivot = tablica[prawy]
i = lewy - 1
dla j od lewy do prawy - 1 wykonaj
jeśli tablica[j] <= pivot wtedy
i = i + 1
zamień(tablica[i], tablica[j])
koniec jeśli
koniec dla
zamień(tablica[i + 1], tablica[prawy])
zwróć i + 1
koniec funkcji

funkcja zamień(a, b)
temp = a
a = b
b = temp
koniec funkcji
```

d) HeapSort (sortowanie przez kopcowanie)

Heapsort, czyli sortowanie przez kopcowanie, to efektywny algorytm sortowania oparty na strukturze danych zwanej kopcem. Kopiec jest rodzajem drzewa binarnego, które spełnia własność kopca - dla maks-kopca każdy rodzic ma wartość większą lub równą wartości swoich dzieci, a dla min-kopca każdy rodzic ma wartość mniejszą lub równą wartości swoich dzieci.

```
funkcja heapsort(tablica)
n = długość(tablicy)
for i od n/2 - 1 do 0 wykonaj
heapify(tablica, n, i)
koniec for

for i od n-1 do 0 wykonaj
zamień(tablica[0], tablica[i])
heapify(tablica, i, 0)
koniec for
koniec funkcji

funkcja heapify(tablica, rozmiarKopca, i)
największy = i
lewy = 2*i + 1
prawy = 2*i + 2

jeśli lewy < rozmiarKopca i tablica[lewy] > tablica[największy] wtedy
największy = lewy
koniec jeśli

jeśli prawy < rozmiarKopca i tablica[prawy] > tablica[największy] wtedy
największy = prawy
koniec jeśli
```

```
        jeśli największy != i wtedy
        zamień(tablica[i], tablica[największy])
        heapify(tablica, rozmiarKopca, największy)
        koniec jeśli
        koniec funkcji

        funkcja zamień(a, b)
        temp = a
        a = b
        b = temp
koniec funkcji
```

Jednym z naszych zadań było zmierzenie czasu pracy algorytmów, tzw. processor time oraz work time. Dodatkowo przeprowadzaliśmy tzw. Unit Test'y każdego z algorytmów.

Processor time - Jest to suma czasu, przez jaki każdy procesor (lub rdzeń procesora) spędza na wykonywaniu danego algorytmu. W przypadku jednowątkowego wykonania algorytmu, czas procesora jest równy czasowi rzeczywistemu (wall-clock time), jaki upłynął od rozpoczęcia do zakończenia algorytmu. W przypadku równoległego wykonania algorytmu przez kilka rdzenii, czas procesora uwzględnia czas pracy wszystkich procesorów. Na przykład, jeśli algorytm działa przez 2 sekundy na 4 rdzeniach, to czas procesora wynosi 8 sekund (2 sekundy * 4 rdzenie).

Work time - Jest to całkowity czas pracy algorytmu, który może być liczony jako czas procesora w kontekście równoległego wykonania algorytmu. Work time w algorytmach równoległych odnosi się do sumarycznego czasu, jaki wszystkie procesory spędzają na wykonaniu algorytmu. Jest to miara całkowitego nakładu pracy, który został wykonany przez wszystkie procesory.

Google Test - popularna biblioteka testowa stworzona przez firmę Google, służąca do pisania i uruchamiania testów jednostkowych w języku C++. Biblioteka ta ułatwia tworzenie testów, organizowanie ich oraz raportowanie wyników, co jest kluczowe dla zapewnienia jakości oprogramowania.

2. ZŁOŻONOŚĆ

Dla MergeSort

Główne operacje: porównanie 2 elementów lub indeksów

Rozmiar danych : 2 tablice – pierwotna oraz tymczasowa

Poziomy głębokości : każdy podział ma kolejne poziomy głębokości

Na każdym poziomie rekursji funkcja merge() jest wywoływana na ciągach o łącznej długości len (najpierw na 2 potem na 4, etc.) Ponieważ ciąg dzielony jest na 2 połowy, więc głębokość (liczba poziomów) rekursji wynosi **Θ(log2(n))**

Zatem, mamy Θ(log2(n)) poziomów rekursji i na każdym poziomie wszystkie wywołania funkcji merge mają łączną złożoność Θ(n).

Podsumowując, otrzymujemy złożoność obliczeniową:

$$\mathbf{W(n) = A(n) = \Theta(n \cdot \log(n))}$$

Dla HeapSort

Algorytm wykonuje dwie główne operację od których jest zależna złożoność:

Budowanie kopca z tablicy o długości n wymaga O(n) operacji. Proces ten polega na przekształceniu tablicy wejściowej w kopiec binarny poprzez wywołanie funkcji heapify dla każdego węzła kopca, począwszy od liści. Dzięki temu, największe (lub najmniejsze, w zależności od rodzaju kopca) elementy znajdują się na szczycie kopca.

Wyodrębnianie każdego elementu z kopca i przywracanie właściwości kopca wymaga przejścia przez 2^h poziomów kopca w każdym kroku. Ponieważ każde piętro jest większe *2 niż poprzednie , czyli – 2^0 = 1 , 2^1 = 2,2^2 =4 , ..., 2^n = log2 n. W algorytmie sortowania przez kopcowanie, wyodrębnianie wszystkich n elementów z kopca wymaga w najgorszym przypadku log n operacji wliczając w to wszystkie przejścia oraz naprawę kopca po wyodrębnieniu. Złożoność wynosi więc:

$$\mathbf{W(n) = \Theta(n \cdot \log(n))}$$

Dla QuickSort

W przypadku optymistycznym, jeśli mamy szczęście za każdym razem wybrac mediane z sortowanego fragmentu tablicy, to liczba porównań niezbędnych do uporządkowania *n*-elementowej tablicy opisana jest rekurencyjnym wzorem:

$$T(n) = (n-1) + 2T\left(\frac{n-1}{2}\right)$$

Co daje finalnie liczbe porównań równą **Θ(n · log(n)).** A więc złożoność naszego algorytmu będzie równa:

$$\mathbf{T(n) = \Theta(n \cdot \log(n))}$$

W przypadku pesymistycznym natomiast, złożoność będzie wynosiła

T (n) = Θ(n^2)

Dzieje się tak, ponieważ w przypadku pesymistycznym zakładamy wybieranie zawsze najmniejszego lub największego elementu w sortowanym fragmencie tablicy.

Dla BubbleSort

Algorytm wykonuje **n - 1** przejść, a w każdym przejściu wykonuje **n - k** porównań (gdzie **k** to numer przejścia), przez co jego teoretyczna złożoność czasowa wynosi **Θ(n^2).** W podstawowej wersji algorytmu nie można tego czasu skrócić, a każda permutacja powoduje, że algorytm jest wykonywany w czasie pesymistycznym.

3. CZAS PRACY

MERGESORT

n = 50000	**106 ms**
n = 1	**0 ms (jakoś kosmicznie mało)**

QUICKSORT

n = 50000	**3.4 ms**
n = 1	**0 ms (jakoś kosmicznie mało)**

BUBBLESORT

n = 50000	**4200 ms**
n = 1	**0 ms (jakoś kosmicznie mało)**

HEAPSORT

n = 50000	**49 ms**
n = 1	**0 ms (jakoś kosmicznie mało)**

4. PODSUMOWANIE, BIBLIOGRAFIA, WNIOSKI

Program został wykonany w oparciu o system kontroli wersji GitHub przy pisaniu kodu. Projekt był tworzony na kilku repozytoriach (z każdą większą aktualizacją kodu, dodawaniem nowych plikow etc. Tworzyliśmy nowe repozytorium, gdyż jeszcze jesteśmy dosyć początkujący w używaniu systemów GIT).

Projekt wykonaliśmy w 3 osobowym zespole z podziałem na zadania:

Dominik Koralik: HeapSort, Menu, twórca repozytoriów

Maciej Jankoś: BubbleSort, ½ QuickSort oraz sporządzenie sprawozdania

Łukasz Konieczny: MergeSort, google test, ½ QuickSort

Zmierzyliśmy czas pracy każdego z algorytmów i doszliśmy do wniosków, że QuickSort oferuje najszybsze posortowanie dużej ilości danych. BubbleSort jest co prawda prosty w implementacji, jednak oferuje najgorsze efekty przy większej grupie danych. Dobrym wyborem może być także HeapSort, który posortował tablice w niewiele większym czasie od QuickSort'u. Neutralnym sortowaniem będzie sortowanie MergeSort, który posortował co prawda jako 3. w kolejności, lecz nadal był to akceptowalny czas.

Dzięki temu projektowi pogłębliśmy wiedze z programowania obiektowego czy z algorytmów sortowania. Również nauczyliśmy się i poszerzyliśmy wiedze o systemie GIT. Poznaliśmy testy jednostkowe i wykorzystaliśmy tzw. google test do ich wykonania w naszym projekcie. Praca w grupie pozwoliła na polepszenie relacji jako drużyna oraz spowodowała że w przyszłości będziemy lepiej pracować jako członkowie grupy. Projekt ten możemy w przyszłości użyć jako uzupełnienie naszego programistycznego portfolio.

BIBLIOGRAFIA

https://www.geeksforgeeks.org/merge-sort/

https://www.geeksforgeeks.org/heap-sort/

https://pl.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort

https://www.geeksforgeeks.org/bubble-sort/

https://learn.microsoft.com/pl-pl/visualstudio/test/how-to-use-google-test-for-cpp?view=vs-2022