|  |  |
| --- | --- |
| Kierunek, rok:  ITE NST, 2024/25 | Imię i nazwisko:  Maciej Lorek,  Krystian Kostrzewa,  Juliusz Czechura |
| Tytuł ćwiczenia:  Proste i złożone algorytmy sortowania | Data laboratoriów:  28.04.2024 |

1. Proste algorytmy sortowania:
   1. Opis kodu:

Na początku programu w funkcji ‘*main’* następuje deklaracja zmiennych ‘*start’, ‘stop’* typu ‘*clock\_t’*, tablicy elementów ‘*int’* i uzupełnienie jej losowymi wartościami funkcją ‘*fill\_table’*.

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

W pętli ‘*for’* losowane są liczby z zakresu 2 razy większego niż rozmiar tablicy i są dodawane, aż do zapełnienia tablicy.

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

W funkcji ‘*main’* sprawa wygląda prosto. Dla każdej metody sortowania najpierw nastąpi skopiowanie ‘*unsorted\_table’* do tablicy ‘*table’* za pomocą funkcji ‘*copy\_table’*. Następnie odbywa się sczytanie czasu przed wykonaniem sortowania i zapisanie go do zmiennej ‘*start’*, następnie wykonuje się algorytm sortowania na tabeli ‘*table’*  i na końcu sczytywany jest czas po wykonaniu funkcji. Po tych operacjach wyświetlana jest informacja na temat czasu wykonania danej funkcji.

Obraz zawierający tekst, Oprogramowanie multimedialne, oprogramowanie, Czcionka

Opis wygenerowany automatycznie

Funkcja ‘*copy\_table’* działa na zasadzie przechodzenia przez tablicę źródłową i nadpisywaniem danymi z niej danych z tablicy docelowej.

Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

Opis wygenerowany automatycznie

Funkcja ‘*bubble\_sort* ‘ jest algorytmem sortowania, który porównuje sąsiednie elementy tablicy i zamienia je miejscami, jeśli są w złej kolejności. Wykonuje to wielokrotnie, przechodząc przez tablicę, aż do momentu, gdy wszystkie elementy są uporządkowane. Jest to jedna z prostszych metod sortowania, choć może być nieefektywna dla dużych zbiorów danych. W implementacji tej funkcji, pętle wewnętrzne iterują po tablicy, porównując elementy i wymieniając je, jeśli to konieczne.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

Funkcja ‘*insert\_sort* ‘ sortuje tablicę liczb całkowitych w rosnącej kolejności. Wykorzystuje ona algorytm ‘sortowania przez wstawianie’, który porównuje elementy tablicy z już posortowaną częścią, przesuwając większe elementy w prawo, aby wstawić aktualnie sprawdzany element w odpowiednie miejsce. Proces ten powtarza się dla każdego elementu tablicy, aż do momentu, gdy cała tablica jest posortowana. W implementacji tej funkcji, dla każdego elementu tablicy iteruje się wstecz od jego pozycji, przesuwając większe elementy o jedno miejsce w prawo, aż do znalezienia odpowiedniej pozycji dla aktualnego elementu.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie

Opis wygenerowany automatycznie

Funkcja ‘*selection\_sort* ‘ sortuje tablicę liczb całkowitych w rosnącej kolejności. Wykorzystuje ona algorytm ‘sortowania przez wybieranie’, który iteruje po tablicy, znajdując najmniejszy element i zamieniając go z pierwszym nieposortowanym elementem. Proces ten powtarza się dla każdego elementu tablicy, tworząc posortowaną sekcję na początku tablicy. W implementacji tej funkcji, dla każdego elementu tablicy, znajduje się najmniejszy element w nieposortowanej części tablicy, a następnie zamienia się go z pierwszym elementem nieposortowanym.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

Funkcja ‘*comb\_sort* ‘ to algorytm sortowania, który opiera się na technice ‘grzebieniowej’. Początkowo ustawia się szerokość odstępu (ang. gap) na rozmiar tablicy. Następnie pętla ‘*while’* iteruje, zmniejszając odstęp w każdej iteracji o około 23%, aż osiągnie wartość 1 lub gdy żadna zamiana nie będzie już potrzebna. W ramach tej pętli, iteruje się po tablicy, porównując elementy oddalone o aktualny odstęp, zamieniając je, jeśli są w złej kolejności. Proces ten powtarza się aż do momentu, gdy cała tablica jest posortowana.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

Funkcja ‘*shell\_sort’* to algorytm sortowania, który opiera się na technice ‘sortowania Shella’. Początkowo ustawia się odstęp (‘gap’) na połowę rozmiaru tablicy, a następnie zmniejsza się go w każdej iteracji o połowę, aż do wartości 1. W ramach pętli zewnętrznej, iteruje się po tablicy, zaczynając od elementów na odstępie gap. Dla każdego elementu, który zostaje przeanalizowany, porównuje się go z elementami w swojej podtablicy na wcześniejszych pozycjach, zamieniając je, jeśli są w złej kolejności. Proces ten kontynuuje się dla coraz mniejszych odstępów, aż do momentu, gdy cała tablica jest posortowana.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

* 1. Pomiary czasu sortowania:
* Tablica 1000 elementów:

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

* Tablica 10 000 elementów:

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

* Tablica 30 000 elementów:

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

* 1. Wnioski:

Przy małych tablicach (1000 elementów) wszystkie algorytmy radziły sobie dobrze. Różnice w pomiarach były niewielkie. Pierwsze zauważalne różnice w długości trwania sortowania obserwujemy tak naprawdę przy średnich tablicach (10000 elementów). Sortowanie bąbelkowe znacznie zwolniło i jest zdecydowanie najwolniejszym algorytmem. Średnio radziły sobie sortowanie poprzez wstawianie i wybieranie (pierwsze nieco lepiej). Najlepiej natomiast poradziły sobie sortowanie grzebieniowe oraz Shella, a różnica w czasie wykonania mimo znacznej różnicy w wielkości tablicy nie zwiększyły się znacząco. Identycznie jak w przypadku tablic średnich sytuacja wygląda jeśli chodzi o wielkie tablice (30000 elementów). Hierarchia algorytmów nie zmieniła się, a różnice pomiędzy ich szybkościami jedynie stały się jeszcze bardziej widoczne.

1. Złożone algorytmy sortowania:
   1. Sortowanie kubełkowe:

Kod przedstawia implementację sortowania kubełkowego dla tablicy liczb całkowitych. Algorytm ten polega na podzieleniu zbioru na "kubełki" odpowiadające określonym zakresom wartości, a następnie sortowaniu każdego kubełka osobno, po czym łączeniu ich w jedną posortowaną tablicę.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

Przypadki użycia sortowania kubełkowego obejmują:

* Sortowanie liczb zmiennoprzecinkowych: Sortowanie kubełkowe jest skuteczne w sortowaniu liczb zmiennoprzecinkowych, ponieważ można łatwo przyporządkować je do odpowiednich kubełków na podstawie ich wartości.
* Równomierna dystrybucja danych: Gdy dane są równomiernie rozłożone na przedziale, sortowanie kubełkowe może być bardzo efektywne i szybkie.
* Dane o ograniczonym zakresie: Kiedy wiadomo, że dane mieszczą się w określonym zakresie wartości, sortowanie kubełkowe może być bardziej efektywne niż inne algorytmy, szczególnie przy większych zbiorach danych.

Tablica 10 000 elementowa, została posortowana w czasie: 0.527 [ms]. Co pozwala na wyciągniecie następujących wniosków:



Algorytm ten może być mniej wydajny w przypadku danych rozproszonych nierównomiernie lub gdy liczba kubełków jest bardzo duża, co może skutkować większym narzutem pamięciowym. Jednak dla odpowiednio dobranego rozmiaru kubełków i odpowiednio rozłożonych danych może być bardzo efektywnym sposobem sortowania.

* 1. Sortowanie niestandardowych danych:

Ten fragment kodu zawiera definicję klasy ‘Person’, która ma pola ‘name’ i ‘age’. Następnie jest zdefiniowana funkcja ‘sortByAge’, która porównuje obiekty typu Person względem ich wieku i sortuje je malejąco według wieku.

Obraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznie

Obraz zawierający tekst, Czcionka, zrzut ekranu, numer

Opis wygenerowany automatycznie

Algorytm sortowania można dostosować do sortowania danych niestandardowych na podstawie określonych kryteriów poprzez zdefiniowanie niestandardowej funkcji porównującej (np. ‘sortByAge’), która określa sposób porównywania obiektów i wyznacza, w którym kierunku mają być posortowane. W przypadku innych kryteriów sortowania (np. imienia, nazwiska, innych atrybutów), należy napisać odpowiednie funkcje porównujące uwzględniające te kryteria.

* 1. Sortowanie cząstkowe (Partial Sort):

Ten kod implementuje algorytm sortowania k-tych największych elementów za pomocą struktury kopca (‘heap’).

Obraz zawierający tekst, zrzut ekranu, oprogramowanie

Opis wygenerowany automatycznie

* Funkcja: ‘heapify(vector<int>& arr, int n, int i)’:

Funkcja ta jest częścią algorytmu sortowania przez kopcowanie (‘heap sort’). ‘arr’ to wektor liczb całkowitych, ‘n’ to wielkość wektora, a ‘i’ to indeks węzła kopca. Funkcja porównuje elementy węzła i z jego lewym i prawym dzieckiem (jeśli istnieją) i wymienia je w odpowiedni sposób, aby zachować własność kopca (największy element w korzeniu).

* Funkcja: ‘partialSortHeap(vector<int>& arr, int k)’:

Ta funkcja sortuje tylko pierwsze ‘k’ największych elementów z wektora ‘arr’. Inicjalizuje zmienną ‘n’ jako wielkość wektora i tworzy pusty wektor ‘result’ na wynik. Wywołuje funkcję ‘heapify’ dla każdego węzła kopca, począwszy od ostatniego węzła wewnętrznego, aby zbudować kopiec. Następnie przesuwa największe elementy na koniec wektora, co jest krokiem sortowania przez kopcowanie. Dodaje ‘k’ ostatnich elementów (największych) do wektora wynikowego ‘result’ i zwraca ten wektor.

Podsumowując, algorytm ten wykorzystuje strukturę kopca do częściowego sortowania wektora, przechowując jedynie k największych elementów. Jest to efektywny sposób sortowania dużych danych, gdy potrzebujemy jedynie częściowo posortowanych wyników.

* 1. Hybrydowy algorytm sortowania:

Ten kod implementuje trzy różne algorytmy sortowania: sortowanie przez wstawianie (‘insertionSort’), sortowanie szybkie (‘quickSort’), oraz sortowanie przez scalanie (‘mergeSort’). Dodatkowo, jest funkcja ‘hybridSort’, która wybiera odpowiedni algorytm sortowania w zależności od rozmiaru wektora wejściowego.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznie

Algorytmy sortowania są wybierane na podstawie długości tablicy wejściowej:

* tablica < 20 - sortowanie przez wstawianie,
* tablica >= 20 && tablica < 1000 - sortowanie szybkie,
* tablica >= 1000 - sortowanie przez scalanie.

Kryteria wyboru algorytmu sortowania w tym przypadku to głównie liczba elementów w tablicy. Sortowanie przez wstawianie jest skuteczne dla małych zbiorów danych, ponieważ ma niską złożoność obliczeniową O(n^2), co jest akceptowalne dla niewielkich danych. Sortowanie szybkie jest wydajne dla średnich rozmiarów danych i ma złożoność O(n log n), podczas gdy sortowanie przez scalanie jest efektywne dla dużych zbiorów danych, również złożoność O(n log n).

Można również rozważyć inne kryteria, takie jak rodzaj danych do posortowania (np. dane losowe, częściowo posortowane, odwrócone), ponieważ różne algorytmy sortowania mogą radzić sobie lepiej w różnych przypadkach. Na przykład, sortowanie szybkie może działać gorzej dla już posortowanych danych, w przeciwieństwie do sortowania przez scalanie, które ma stałą złożoność niezależnie od rodzaju danych. Oczywiście, takie kryteria wymagałyby bardziej zaawansowanej analizy i testowania dla konkretnych przypadków danych.

* 1. Sortowanie w miejscu (In-Place Sorting):

Ten kod zawiera implementacje dwóch algorytmów sortowania: sortowanie przez wstawianie (‘insertion sort’) oraz sortowanie przez scalanie (‘merge sort’). W funkcji ‘compareSortTimes’ obie metody sortowania są porównywane pod względem czasu wykonania na tych samych danych wejściowych.

Obraz zawierający tekst, zrzut ekranu, wyświetlacz, Czcionka

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Czcionka

Opis wygenerowany automatycznie

Jeśli chodzi o algorytmy sortowania w miejscu (bez dodatkowej pamięci), to sortowanie przez wstawianie jest jednym z przykładów. W przypadku sortowania przez scalanie, które wymaga dodatkowej pamięci na pomocniczą tablicę podczas scalania, ten dodatkowy krok może wpłynąć na czas wykonania. Sortowanie przez wstawianie działa szybciej niż sortowanie przez scalanie, co może wynikać m.in. z tego, że sortowanie przez wstawianie nie wymaga dodatkowej pamięci na skalę znacząco jak sortowanie przez scalanie, gdzie używamy dodatkowej tablicy do scalania części tablicy. Jednakże, poniższy test został wykonany dla tablicy 10 000 elementowej. Jeśli ilość danych do posortowania byłaby większa to sortowanie przez scalanie może być bardziej efektywne, zwłaszcza jeśli zastosujemy optymalizacje (np. iteracyjną wersję sortowania przez scalanie).

