Laboratorium 2

Badanie złożoności obliczeniowej algorytmów na przykładzie sortowania

Cel ćwiczenia

Celem ćwiczenia jest praktyczne zapoznanie się z pojęciem złożoności obliczeniowej na przykładzie wybranych algorytmów sortowania: bąbelkowego, przez selekcję, przez wstawianie, przez scalanie, Shella oraz sortowania szybkiego. Dodatkowo badana może być stabilność i wykorzystanie stosu systemowego przez algorytmy. Do badań tych wykorzystywany jest program aisd23, opisany poniżej. Sortowane są rekordy umieszczone w tablicy dostępnej przez wskaźnik *array*. Standardowo do dyspozycji są rekordy o jednym lub o dwóch kluczach zawierających liczby całkowite (z powtórzeniami lub bez). Mogą to być liczby wylosowane z rozkładu równomiernego lub ciągi liczb uporządkowane rosnąco lub malejąco.

W ramach wykonywania ćwiczeń studenci mogą również tworzyć własne typy rekordów. Typy te muszą dziedziczyć po klasie *Object*, w której znajduje się kilka metod czysto wirtualnych, więc klasy nowych rekordów powinny zdefiniować ciała tych metod. Tablicę własnych rekordów należy alokować w pliku *main.c* według znajdujących się tam wzorów. Nie ma przy tym potrzeby modyfikowania implementacji algorytmów (pod warunkiem poprawnego zdefiniowania własnych rekordów).

UWAGA! Do porównania kilku metod sortowania powinny być wykorzystywane zawsze **te same dane wejściowe**. Wszystkie pomiary czasu powinny być wykonywane na tej samej maszynie a przynajmniej na maszynach o identycznych parametrach, (np. na dwóch różnych komputerach laboratoryjnych). Należy wziąć to pod uwagę planując wykonanie części ćwiczenia w domu.

Struktura programu *aisd23*

Program *aisd23* jest wykorzystywany w dwóch ćwiczeniach: drugim (sortowanie metodami „standardowymi”) i trzecim (sortowanie z użyciem drzew binarnych oraz inne zastosowania takich drzew). W niniejszej instrukcji przedstawiona zostanie tylko ta część programu, której używa się w ćwiczeniu nr 2 (jak również część wspólna dla obu ćwiczeń).

Program *aisd23* zawiera implementacje sporej części omawianych na wykładzie metod sortowania, przy czym niektóre z nich występują tu w kilku wersjach. Program może wypisywać na konsolę następujące informacje:

- zawartość tablicy rekordów na wejściu,
- zawartość tablicy rekordów po posortowaniu,
- czas działania części programu wykonującej algorytm sortowania (w sekundach),
- liczby operacji przeprowadzonych w trakcie wykonywania algorytmu:

  - liczba kopiowań rekordów (uwaga: standardowa zamiana dwóch rekordów miejscami wymaga trzech kopiowań; ogólnie: „przesuwanie" liczby N sąsiadujących rekordów z użyciem bufora wymaga $N + 2$ kopiowań),
  - liczba porównań kluczy,
- liczby potrzebnych do wykonania algorytmu operacji na indeksach tablicy (iteratorach):
  - liczba porównań (np. sprawdzanie czy osiągnięty został początek lub koniec tablicy),
  - dodawań/odejmowań (przesuwanie iteratora),
  - mnożeń („skoki" iteratora),
  - dzieleń („skoki" iteratora),
- maksymalna głębokość stosu systemowego – wyłącznie w funkcjach zaimplementowanych rekurencyjnie.

Kod programu jest rozbity na wiele plików lecz wykonanie ćwiczenia wymaga wprowadzenia do nielicznych z nich zaledwie kilku modyfikacji. **Po wprowadzeniu dowolnej zmiany w kodzie należy usunąć poprzednie produkty kompilacji i ponownie skompilować cały projekt, wpisując kolejno *make clean* i *make*.** Poniżej wymienione są najważniejsze pliki, które mogą (lub powinny) być modyfikowane w ćwiczeniu.

1. **Podstawowe paramenty eksperymentu (niezależne od badanego algorytmu)** – plik *control.h.* W pliku tym zdefiniowane są m.in.:
   - NUMER_INDEKSU – ziarno generatora liczb pseudolosowych. Wykonywanie ćwiczenia należy rozpocząć od nadania mu wartości równej **swojemu numerowi indeksu**. **Należy pamiętać, że kolejne uruchomienia programu z niezmienionym ziarnem dają identyczne sekwencje pseudolosowe.** Ten efekt jest pożądany, gdy chcemy mieć pewność, że porównywane algorytmy sortują dokładnie ten sam ciąg danych. Możliwe jest również inicjowanie generatora liczb pseudolosowych na podstawie bieżącego czasu. Przydaje się to np. w przypadku wielokrotnego wywoływania tego samego algorytmu dla wielu różnych ciągów wejściowych o identycznej długości w celu wykonania badań statystycznych.
   - CWICZENIE_2 – flaga pozwalająca wykonywać eksperymenty przewidziane dla ćwiczenia 2 lub ćwiczenia nr 3.
   - LICZNOSC – liczność badanego zbioru danych. Sensowna wartość oraz zakres zmian wartości tego parametru są silnie uzależnione od złożoności badanego algorytmu.
   - CIAG_WEJSCIOWY_ROSNACY, CIAG_WEJSCIOWY_MALEJACY – ustawienie (nadanie wartości 1) **jednej** z tych flag powoduje wypełnienie tablicy N-elementowej rekordami o kluczach odpowiednio rosnących lub malejących. Dla dostępnych standardowo kluczy będących liczbami całkowitymi będzie to ciąg 0, 1, 2, …, N–1 lub ciąg N–1, N–2, …, 0. Jeśli **obie** flagi są zerami tablica jest wypełniana rekordami o kluczach generowanych pseudolosowo z wartościami mającymi rozkład równomierny (dla liczb całkowitych z przedziału [0, N–1]).
   - REKORD_Z_JEDNYM_KLUCZEM – flaga określająca liczbę kluczy w rekordzie (jeden lub dwa). W trakcie działania programu wybór klucza do sortowania przeprowadzany jest procedu-

rą *Object::setKeyID*(*0*) (lub *Object::setKeyID*(*1*)) i obowiązuje wszystkie algorytmy aż do następnego wyboru.

- KLUCZE_UNIKATOWE – flaga określająca czy wartości kluczy w badanym zbiorze rekordów mogą się powtarzać (używana tylko w przypadku wypełnienia tablicy rekordami z kluczami generowanymi pseudolosowo).
- MAX_WARTOSC_KLUCZA – stosowana wyłącznie dla kluczy całkowitoliczbowych, ignorowana przy ustawieniu flagi KLUCZE_UNIKATOWE na tak. Powinna być >= 0, z tym, że dla wartości 0 MAX_WARTOSC_KLUCZA ustawiana jest standrdowo na LICZNOSC–1.
- ALGORYTM_RAPORT i IMPLEMENTACJA_RAPORT – flagi decydujące o przeprowadzaniu zliczania kluczowych operacji (patrz powyższy punkt *Struktura programu*). Należy zdawać sobie sprawę, że **samo zliczanie operacji jest również operacją trwającą pewną liczbę kroków maszynowych, a zatem zwiększającą czas wykonywania programu.** W zależności od tego co jest celem danego zadania w ćwiczeniu należy zrezygnować z rejestrowania nieistotnych operacji nadając odpowiednim makrodefinicjom wartość 0.

2. **Wybór algorytmu sortowania** – plik *badanie_alg_sort-std.hpp*. Należy badać tylko jeden algorytm jednocześnie (pozostałe komendy wywołujące inne algorytmy powinny być wykomentowane). Zaimplementowane (lub zadeklarowane) zostały następujące metody sortowania:
   - **Selekcja** (wybór) – w dwóch wersjach.
   - **Wstawianie** – z dwiema metodami wyboru pozycji, na którą jest wstawiany rekord.
   - **Shella** – z trzema dostępnymi ciągami sterującymi oraz z możliwością implementacji własnego ciągu. Sortowanie Shella wykorzystuje sortowanie przez wstawianie zatem należy pamiętać, że zmienianie ustawień (flag) dla tego sortowania (patrz punkt 3) będzie wpływać na wykonywanie sortowania Shella.
   - **Sortowanie szybkie** – w najpopularniejszej wersji zaproponowanej przez Hoare'a (podanej na wykładzie) oraz w wersji Lomuto (opisanej np. w książce T. Cormen et al. „Wprowadzenie do algorytmów"). Ta pierwsza wersja występuje tu w trzech wariantach, są to warianty:
     - podstawowy,
     - z przełączaniem na sortowanie przez wstawianie jeśli analizowany podciąg rekordów jest nie dłuższy niż podany argument (*factor*),
     - jak w poprzednim pod punkcie ale dodatkowo z elementem rozdzielającym (ang. *pivot*) definiowanym jako mediana z określonej liczby ostatnich elementów analizowanego ciągu.
   - **Bąbelkowe** i **koktajlowe** (ang. *shakersort*) oraz **przez scalanie** – w programie stworzono jedynie interfejsy funkcji dokonujących sortowanie (ciała funkcji do ewentualnej samodzielnej implementacji algorytmów przez wykonującego ćwiczenie).

   W omawianym pliku można ponadto decydować o wyprowadzaniu zawartości rekordów na konsolę przez odkomentowanie/wykomentowanie funkcji *wypiszNaKonsole*() (dla predefiniowanych standardowych rekordów wyprowadzane są wyłącznie wartości kluczy). Znajomość zawartości tablicy rekordów pozwala **ocenić poprawność działania algorytmu (co jest konieczne w zadaniach wymagających modyfikacji kodu)**. Poza tym dla danych charakteryzujących się kilkoma

kluczami można w ten sposób oszacować stabilność algorytmu. Uwaga: w przypadku dużych zbiorów danych sugeruje się wyłączenie wyświetlania.

3. **Parametry poszczególnych algorytmów** – plik *control_sort-std.h*. Zdefiniowane są tu makrodefinicje pozwalające sterować przebiegiem niektórych algorytmów. Należy mieć świadomość, że pewne algorytmy (lub ich fragmenty) są wykorzystywane przez inne, bardziej złożone algorytmy, przez co ustawienie flag dla jednych algorytmów może mieć również wpływ na przebieg działania innych. Poniżej lista dostępnych makrodefinicji.
   - SORTOWANIE_W_MIEJSCU – należy ją wyzerować, jeśli planujemy implementację algorytmu, który nie sortuje „w miejscu". Dzięki temu otrzymujemy do dyspozycji dodatkową tablicę o identycznej wielkości, jak oryginalna tablica danych. Dodatkowa tablica dostępna jest przez wskaźnik *arrayBis*.
   - SELEKCJA_START_OD_PRAWEJ – ustawiana gdy algorytm przez selekcję ma rozpoczynać działanie od umieszczania na końcu tablicy rekordów o największych kluczach. Domyślnie flaga ta jest wyzerowana i algorytm rozpoczyna działanie od umieszczenia kolejnych rekordów o najmniejszych kluczach na początku tablicy (tak, jak na wykładzie i w większości źródeł literaturowych). Proszę zwrócić uwagę, że w obu przypadkach otrzymujemy ciągi rekordów o kluczach rosnących.
   - SELEKCJA_INDEX – gdy ta flaga jest ustawiona, algorytm działa jak klasyczna selekcja znana z literatury. W przeciwnym przypadku podczas przeglądania nieposortowanej części tablicy każdy rekord o wartości klucza mniejszej od wartości klucza pierwszego rekordu (w wariancie SELEKCJA_START_OD_PRAWEJ – większej od wartości klucza ostatniego rekordu) jest z nim na bieżąco zamieniany.
   - WSTAWIANIE_START_OD_PRAWEJ – flaga dla sortowania przez wstawianie, której znaczenie jest analogiczne do znaczenia flagi SELEKCJA_START_OD_PRAWEJ. Należy zwrócić uwagę, że wstawianie jest także częścią składową algorytmów Shella i sortowania szybkiego, więc wartość tej flagi może mieć wpływ na parametry wydajnościowe implementacji tych algorytmów.
   - WSTAWIANIE_SHELL – flaga jest ustawiana jeśli sortowanie ma się odbywać według algorytmu Shella, przy czym elementarne *h*-zbiory są sortowane przez wstawianie. Wyzerowanie flagi oznacza wykonywanie „zwykłego" sortowania przez wstawianie.
   - WSTAWIANIE_BISEKCJA – wybór metody wyszukiwania pozycji, na którą w algorytmie sortowania przez wstawianie (lub Shella) ma trafić rekord o danym kluczu. Dostępne metody to bisekcja i przegląd bezpośredni.

Przebieg ćwiczenia

Dla wskazanych przez prowadzącego algorytmów sortowania należy przystosować kod przykładowego programu do zaplanowanych pomiarów i symulacji. W zadaniach wymagających modyfikacji

algorytmów **konieczna jest weryfikacja poprawności działania własnych metod i zamieszczenie w sprawozdaniu zmodyfikowanego kodu.**

Przykładowe zadania do wykonania na ćwiczeniu (prowadzący może podać inne):

1. Zbadać złożoność obliczeniową wskazanych algorytmów sortowania wziąwszy pod uwagę początkowe uporządkowanie sortowanego zbioru danych.
2. Wskazać, zaimplementować i przetestować możliwe rozwiązania prowadzące do poprawy wydajności badanych algorytmów sortowania.
3. Określić stabilność wskazanych algorytmów sortowania. Przeanalizować możliwość stabilizacji badanych w ćwiczeniu niestabilnych algorytmów sortowania.
4. Napisać procedury generujące różne rozkłady danych do sortowania. Następnie sprawdzić jak wydajność wybranych algorytmów sortowania zależy od rozkładu danych wejściowych.
5. Określić zależność wydajności algorytmu sortowania metodą Shella od rodzaju zastosowanego ciągu **sterującego**. Oprócz trzech ciągów już używanych w programie należy zaimplementować przynajmniej jeden dodatkowy: znany z literatury lub własny. W tym celu wystarczy stworzyć własną klasę dziedziczącą po klasie *SequenceGenerator* (plik *algorytmy_sort_std/shell/generator.h*) i zaimplementować w niej funkcję wirtualną *nextHvalue*. Można tu wykorzystać dostępny zaczątek takiej klasy – *CustomSequenceGenerator*.
6. Jak wiadomo funkcja sortowania szybkiego wywołuje się rekurencyjnie dla coraz mniejszych fragmentów zbioru wejściowego aż do osiągnięcia zbioru jednoelementowego. Dla bardzo małych zbiorów sortowanie szybkie nie jest jednak najwydajniejszym algorytmem. Dlatego możliwe jest użycie innego algorytmu sortowania w każdym wywołaniu, w którym sortowany ma być odpowiednio mały zbiór. Umożliwiają to wersje algorytmu opatrzone przyrostkiem *Factor*. Ostatni parametr (*factor*) wywołania określa liczność zbioru, poniżej której używane jest już sortowanie przez wstawianie. Należy eksperymentalnie dobrać optymalną wartość *factor* (uwzględniając także wartość zerową, tj. klasyczne sortowanie szybkie) dla kilku różnych wielkości zbioru wejściowego.
7. Jak w poprzednim punkcie z tą różnicą, że jeżeli podzbiór wejściowy ma liczność mniejszą od *factor* natępuje od razu powrót na wyższy poziom rekurencji. Po zakończeniu działania tak zmodyfikowanej funkcji *quickSortHoareFactor* cała tablica pozostaje więc wciąż nieposortowana (choć jest podzielona na odcinki, z których każdy zawiera elementy większe od wszystkich elementów odcinka poprzedniego). Ostatnim krokiem zatem musi być więc posortowanie całej tablicy dodatkowo innym algorytmem (np. przez wstawianie). Należy zaimplementować taką wersję procedury sortowania i sprawdzić poprawność implementacji sortując niewielką tablicę. Następnie należy porównać wydajność tej metody z sortowaniem szybkim w wersji najprostszej oraz w oryginalnej wersji *quickSortHoareFactor* dla różnych wielkości zbiorów danych i różnych wartości *factor*.
8. Zaimplementować algorytm sortowania bąbelkowego i/lub koktajlowego (ang. *shakersort*) i porównać go pod względem wydajności z pozostałymi algorytmami o złożoności kwadratowej.

Badania przeprowadzić dla różnych wielkości zbioru sortowanego przy różnym wstępnym uporządkowaniu danych.

Analiza wyników, wnioski, sprawozdanie

Na ocenę z ćwiczenia bezpośrednio wpływa zawartość oddanego w terminie sprawozdania (wyłącznie w postaci elektronicznej w formacie PDF). Sprawozdanie powinno zawierać następujące elementy:

1. Cel ćwiczenia (własnymi słowami).
2. Stosowne wykresy – wykresy przedstawiające czas wykonania różnych algorytmów w funkcji wielkości zbioru sortowanego, wykresy dynamiczne sortowania (patrz ostatni punkt **Uwag końcowych** i Rys. 1.) itp.,
3. W przypadku zadań wymagających modyfikacji kodu:
   - listing zmienionej funkcji (**z wyraźnym wyróżnieniem własnych modyfikacji**)
   - opis zmian i ich uzasadnienie,
   - wynik **testowania poprawności** własnej implementacji.
4. Podsumowanie ćwiczenia czyli wnioski: **konkretne**, **na temat**, **samodzielne**, **twórcze**.

Sprawozdanie powinno zostać nadesłane pocztą elektroniczną w terminie podanym przez prowadzącego zajęcia (zazwyczaj 7–14 dni). **Pod rygorem nie sprawdzania sprawozdania dodatkowo niezbędne jest wypełnienie następujących wymogów formalnych**:

- jedyny przyjmowany format to **PDF**, przy czym tekst **nie może** być wklejony jako rysunek,
- nazwa pliku zawierającego sprawozdanie musi mieć formę **AISDE_2_Nazwisko_Imię.pdf**.
- temat maila ze sprawozdaniem musi pasować do wzorca: **AISDE_2_Nazwisko_Imię.**

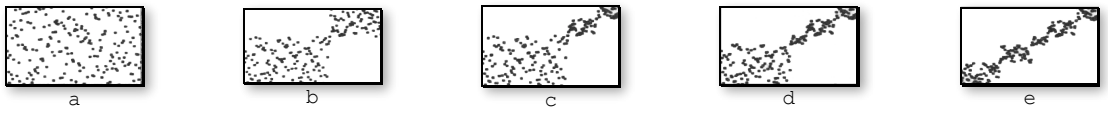
Wymagania do kolokwium wstępnego:

- znajomość badanych algorytmów sortowania,
- znajomość programu *aisd23* (pełne kody źródłowe zamieszczone są na internetowej stronie przedmiotu AISDE http://vlsi.imio.pw.edu.pl/aisde).

Uwagi końcowe

- Dla zapewnienia identycznych warunków pracy algorytmów sortujących w ramach danej serii pomiarowej do sortowania różnymi metodami powinny być wykorzystywane w zawsze **te same dane wejściowe**. Wszystkie pomiary czasu powinny być wykonywane na tej samej maszynie a przynajmniej na maszynach o identycznych parametrach – np. na dwóch różnych komputerach laboratoryjnych. Należy wziąć to pod uwagę, planując wykonanie części ćwiczenia w domu.

- Jednym z możliwych sposobów wizualizacji procesu sortowania są wykresy dynamiczne. Na osi rzędnych znajdują się wartości kluczy, zaś na osi odciętych – ich pozycje. Wykresy dynamiczne dla kilku początkowych etapów sortowania szybkiego przedstawiono na Rys. 1.



Rys. 1. Wykresy dynamiczne sortowania metodą *QuickSort*

Literatura

1. R. Sedgewick *Algorytmy w C++*, Wydawnictwo RM 1999
2. A. Drozdek *C++ algorytmy i struktury danych*, Wydawnictwo Helion 2004
3. T. Cormen et al. *Wprowadzenie do algorytmów*, Wyd. Naukowe PWN 2013
4. J. Grębosz *Symfonia C++* Wydawnictwo Oficyna Kallimach 1997
5. J. Grębosz *Pasja C++* Wydawnictwo Oficyna Kallimach 1997

Opracowanie: dr inż. Dominik Kasprowicz, *dkasprow@imio.pw.edu.pl*
dr inż. Adam Wojtasik, *aw@imio.pw.edu.pl*

NOTATNIK

AISDE LAB 2

Imię i nazwisko, grupa dziekańska	Nr Indeksu	Data wykonania ćwiczenia	Data oddania protokołu

Punkty do wykonania	1	2	3	4	5	6	7	8	9	10	11

Algorytmy do zbadania

- ☐ Sortowanie przez selekcję
- ☐ Sortowanie przez wstawianie
- ☐ Sortowanie Shella
- ☐ Sortowanie bąbelkowe
- ☐ Sortowanie koktajlowe (*shakersort*)
- Sortowanie przez scalanie: ☐ zstępujące ☐ wstępujące
- Sortowanie szybkie: ☐ Hoare ☐ Lomuto ☐ mediana ☐ factor

Algorytm sortowania	Stabilność	Złożoność	Największa liczność zbioru testującego	Uwagi

NOTATNIK NALEŻY ODDAĆ PROWADZĄCEMU ZAJĘCIA W CHWILI ZAKOŃCZENIA ĆWICZENIA