Algorytmy i Złożoność Obliczeniowa

Projekt 1 – (11 marca 2025)

mgr inż. Damian Mroziński

Zakres projektu
C/C++, algorytmy sortowania, analiza wyników, praca na plikach, skryptowanie

Jakie są cele projektu?

- Zapoznanie z różnymi algorytmami sortowania, ich implementacja oraz analiza efektywności.
- Nabycie umiejętności dopasowania programu do wymagań / istniejących klas.
- Nauczenie się podstawowego skryptowania.
- Mądre planowanie projektu, aby nie komplikować sobie życia.

TL;DR
Należy samodzielnie zaimplementować i przeprowadzić analizę określonych algorytmów sortowania (zależnych od planowanej oceny). Jako sortowanie rozumie się uporządkowanie elementów rosnąco. Nie można używać gotowych rozwiązań, również w zakresie struktur danych (typu wektor, lista). Jeśli takie rozwiązania są potrzebne, trzeba je stworzyć samemu, od zera.

Obowiązują następujące założenia

1. Podstawowym elementem sortowanych struktur jest 4-bajtowa liczba całkowitoliczbowa ze znakiem, tj. `int`. Dla badania wpływu typów danych należy rozpatrzyć inne np. `char`, `float`, `double`, lub inne dziwne rzeczy zaakceptowane przez prowadzącego. Jeśli macie inny pomysł, co można sprawdzić - zapraszam do kontatku.
2. Wszystkie sortowane struktury powinny być alokowane i zwalniane dynamicznie (zgodnie z badanym rozmiarem tablicy).
3. Program musi kompilować się **bez ostrzeżen** (tak, ostrzeżeń!).
4. Program musi przejść test debuggera, tj. nie mieć wycieków pamięci i innych błędów.
5. Należy przeprowadzić weryfikację poprawności sortowania. Najlepiej w formie prostej funkcji, która sprawdzi, czy elementy są rosnące (dla małych zbiorów można zrobić wizualnie).
6. Należy zmierzyć czas sortowania tablic (w milisekundach). Mierzony czas dotyczy wyłącznie sortowania i nie wlicza się do niego ładowanie danych z pliku, losowanie wartości, czy zapis wyników do pliku.

7. Pojedynczy pomiar jest niemiarodajny. Aby badania zostały wykonane prawidłowo, należy wywołać algorytm wielokrotnie (np 100 razy), a wyniki uśrednić. Wyniki, które należy przedstawić w sprawozdaniu to w/w średnia, ale również minimum oraz maksimum czasu sortowania.

8. **BADANIE 1)** Dla każdego algorytmu należy przebadać kilka przypadków pod względem liczebności zbioru sortowanego (5 przypadków, na przykład 10,20,30,40,50-tysięcy lub 10,20,40,80,160-tysięcy elementów, w zależności od możliwości sprzętu).

9. **BADANIE 2)** Dla każdego algorytmu należy przebadać wybrany przypadek ze względu na początkowy rozkład elementów (wartości losowe, posortowane malejąco, rosnąco, częściowo - w 33% i 66%).

10. **BADANIE 3)** Dla wybranego przez siebie algorytmu i dla wybranego w tym algorytmie przypadku należy przebadać wpływ typu danych (tj. podstawowy `int` i dwóch innych).

11. Należy zachować spójność badań, aby można było sensownie porównywać wyniki.

12. Przy badaniach należy losować wartości z pełnego zakresu wybranego typu.

13. Kod musi być sformatowany spójnie i zawierać komentarze.

14. Do pomiaru czasu należy zaprojektować klasę `Timer`, zgodnie z podanym plikiem nagłówkowym. Publiczna część musi być dokładnie taka sama, nie większa i nie mniejsza.

```
1  class Timer
2  {
3    public:
4    Timer();       // Initialize and prepare to start.
5    void reset();  // Reset timer.
6    int start();   // Start timer.
7    int stop();    // Stop timer.
8    int result();  // Return elapsed time [ms].
9
10   private:
11   // Everything else you need, both fields and methods.
12 }
```

Zachowując zadany nagłówek, można łatwo podmienić plik klasy, np. zastąpić implementację pomiaru dla Windowsa wersją dla Linuxa i ponownie skompilować program. Jeśli publiczna część nagłówka pozostaje bez zmian, wszystko zadziała poprawnie.

15. Program powinien wykorzystywać szablony, aby łatwo można było wykorzystać zaimplementowane algorytmy do sortowania struktur zawierających elementy różnych typów. **NAPRAWDĘ WARTO!**. Wymagane od 5.0 w górę, ale przyda się każdemu.

16. Plik z danymi wejściowymi zawsze jest zbudowany w ten sam sposób. W pierwszej linii znajduje się liczba danych do odczytu (np 10), a kolejne linie (w tym przypadku 10 kolejnych linii), zawiera wartości, które należy wczytać do programu. W analogiczny sposób należy zapisać posortowane wartości do pliku.

17. Program musi mieć możliwość zapisywania rozwiązania do pliku wyjściowego, w takim samym formacie jak w przypadku pliku wejściowego (liczba elementów + posortowane elementy)

18. Sterowanie programem odbywa się za pomocą argumentów do metody głównej. Należy wprowadzić dwa tryby działania (no, technicznie trzy). Są to: tryb pojedynczego testu, gdzie danymi wejściowymi jest plik, a także tryb badań, gdzie podajemy zakres badanych wartości, a program losuje je przed sortowaniem. Można użyć tego helpa jako bazy i dostosować go pod siebie. Strona pomocy musi ściśle odpowiadać temu, czego program będzie od nas oczekiwał.

```
FILE TEST MODE:
    Usage:
      ./YourProject --file <algorithm> <type> <inputFile> [outputFile]

    <algorithm>   Sorting algorithm to use (e.g., 0 - Bubble, 1 - Merge, ...).
    <type>        Data type to load (e.g., 0 - int, 1 - float).
    <inputFile>   Input file containing the data to be sorted.
    [outputFile]  If provided, the sorted values will be saved to this file.

BENCHMARK MODE:
    Usage:
      ./YourProject --test <algorithm> <type> <size> <outputFile>

    <algorithm>   Sorting algorithm to use (e.g., 0 - Bubble, 1 - Merge, ...).
    <type>        Data type to generate (e.g., 0 - int, 1 - float).
    <size>        Number of elements to generate (instance size).
    <outputFile>  File where the benchmark results should be saved.

HELP MODE:
    Usage:
      ./YourProject --help

    Displays this help message.

    Notes:
    - The help message will also appear if no arguments are provided.
    - Ensure that either --file or --test mode is specified;
      they are mutually exclusive.
```

```
EXAMPLE CALLS:
    Sorting integers using Merge Sort from a file and saving results:
      ./YourProject --file 1 0 input.txt sorted_output.txt

    Running a benchmark with 1000 randomly generated floats using Bubble Sort:
      ./YourProject --test 0 1 1000 results.txt
```

Ocenianie - wymagane algorytmy i dodatkowe zadania

3.0 Sortowanie bąbelkowe, przez scalanie i przez wstawianie. Obiektowość nie jest wymagana (ale bez tego będzie moim zdaniem trudniej).

Od tego momentu trzeba obiektowo

4.0 Sortowanie przez wstawianie (zwykłe i binarne), przez kopcowanie i szybkie. Wyniki wszystkich badań uwzględniają dodatkowo medianę.

4.5 Wymagania jak na 4.0. Jeden algorytm musi mieć dodatkowo wersję pijanego studenta. Badania algorytmu w wersji pijanego studenta są nieco zmodyfikowane. Dla **badania 1)** należy wybrać liczebność na podstawie wyników wersji "podstawowej", a badania przeprowadzić sprawdzając jak parametr pijaństwa wpływa na wyniki (dla pięciu różnych wartości). Dla **2)** należy wybrać liczebność i parametr pijaństwa z punktów poprzednich (reszta bez zmian). Jeśli dla **3)** zostanie wybrany pijany student, należy zastosować zmiany jak w punkcie **2)** (reszta bez zmian). Porównać jak bardzo wersja pijana psuje wyniki (a może wcale nie).

5.0 Sortowanie przez wstawianie, przez kopcowanie, shella i szybki. Jeden algorytm musi mieć dodatkowo wersję pijanego studenta (zgodnie z opisem jak w 4.5). Dodatkowo wyniki wszystkich badań uwzględniają odchylenie standardowe. Szablony są wymagane.

W Badaniu 1), w przypadku algorytmu shella i szybkiego, należy sprawdzić dodatkowe parametry. W przypadku shella badanie rozszerzy się o wpływ wyboru wielkości odstępu (porównać min. dwa zaproponowane wzory), a w przypadku szybkiego o sposób wyboru pivota (skrajny lewy, prawy, środkowy, losowy). Badania nadal powtarzamy dla 5 wielkości instancji, zgodnie z założeniami. Zwycięskich opcji tych dwóch algorytmów należy użyć w badaniach **1-3)**.

5.5 Wszystko jak dla 5.0. **Badanie 3)** zostaje rozszerzone o porównywanie nie trzech typów danych, a czterech. Będą to: `int`, inny wybrany typ prosty np. `float`, łańcuchy znaków oraz... planszówki.

Planszówki definiuje się za pomocą ich nazwy (max 25 znaków), wydawcy (max 25 znaków), minimalnej i maksymalnej liczby graczy (1-10), długości czasu rozgrywki w minutach (5-480min), poziomu skomplikowania (1-10) oraz radości jaką sprawia (1-10). Planszówka jest najlepsza, gdy poziom jej średniej fajności obliczanej na minutę na gracza jest najwyższa. Należy zaproponować wzór który wyliczy taką fajność z uwzględnieniem wszystkich parametrów liczbowych. Może również uwzględniać rzeczy kompletnie głupie typu długość tytułu czy liczbę literek "a" w nazwie wydawcy. A może jeszcze coś innego. Planszówki muszą być

możliwe do wczytywania i zapisywania do pliku w podobnej formie jak liczby (jeden element, jedna linijka).

Planszówki mają być wprowadzone w taki sposób, aby nie trzeba było w żaden sposób modyfikować kodu algorytmów. Wystarczy zmodyfikować szablon.

W razie pytań, pozostaję do dyspozycji.