Algorytmy i struktury danych
Laboratorium 2

Termin oddania: 23 marzec, 6:52

Zadanie 1. [35%]
Celem zadania jest zaimplementowanie i przetestowanie następujących algorytmów sortowania:

- InsertionSort,
- MergeSort,
- QuickSort.

Program przyjmuje dwa parametry wejściowe `--type insert|merge|quick`— określający wykorzystywany algorytm sortowania oraz `--comp '>='|'<='`—określający porządek sortowania.
Wejściem dla programu są kolejno:

- liczba n — długość sortowanej tablicy,
- tablica elementów do posortowania (niech elementy tej tablicy zostaną nazwane kluczami).

Program powinien sortować tablicę wybranym algorytmem i wypisywać na standardowym wyjściu błędów wykonywane operacje (porównania, przestawienia). Po zakończeniu sortowania, na standardowym wyjściu błędów powinna zostać wypisana łączna liczba porównań między kluczami, łączna liczba przestawień kluczy oraz czas działania algorytmu sortującego. Finalnie, program sprawdza, czy wynikowy ciąg jest posortowany zgodnie z wybranym porządkiem, a następnie wypisuje na standardowe wyjście liczbę posortowanych elementów oraz posortowaną tablicę.
Przykładowe wywołanie:

```
./main --type quick --comp '>='
5
9 1 -7 1000 4
```

Zadanie 2. [20%]
Uzupełnij program z **Zadania 1.** o możliwość wywołania go z dodatkowym parametrem uruchomienia `--stat nazwa_pliku k`, wtedy pomija on wczytywanie danych i dla każdego $n \in \{100, 200, 300, \ldots, 10000\}$ wykonuje po `k` niezależnych powtórzeń:

- generowania losowej tablicy n elementowej (zadbaj o dobry generator pseudolosowy),
- sortowania kopii wygenerowanej tablicy,
- dla każdego z sortowań, zapisania do pliku `nazwa_pliku` statystyk odnośnie rozmiaru danych n, liczby wykonanych porównań między kluczami, liczby przestawień kluczy oraz czasu działania algorytmu sortującego.

Po zakończeniu programu, korzystając z zebranych danych przedstaw na wykresach, za pomocą wybranego narzędzia (np. numpy, Matlab, Mathematica):

- średnią liczbę wykonanych porównań kluczy (c) w zależności od n,
- średnią liczbę przestawień kluczy (s) w zależności od n,
- średni czas działania algorytmu w zależności od n,
- iloraz $\frac{c}{n}$ w zależności od n,
- iloraz $\frac{s}{n}$ w zależności od n.

Zadbaj o to, by dane dotyczące różnych algorytmów sortujących można było nakładać na te same osie i porównywać. Sprawdź, jak wykresy zmieniają się dla różnych `k` (np. $\texttt{k} = 1$, $\texttt{k} = 10$, $\texttt{k} = 1000$).

Zadanie 3. [35%]
Uzupełnij **Zadanie 1.** o algorytm Dual-pivot QuickSort używając strategii Count:

- Mamy dwa pivoty p i q oraz załóżmy, że $p < q$.
- Załóżmy, że w procedurze `partition` klasyfikując i-ty element tablicy mamy s_{i-1} elementów małych (mniejszych od p) oraz l_{i-1} elementów dużych (większych od q).
- Jeśli $l_{i-1} > s_{i-1}$: porównuj i-ty element w pierwszej kolejności z q, a następnie, jeśli jest taka potrzeba, z p.
- Jeśli $l_{i-1} \leq s_{i-1}$: porównuj i-ty element w pierwszej kolejności z p, a następnie jeśli jest taka potrzeba, z q.

Dokonaj szczegółowych porównań otrzymanych statystyk dla algorytmu QuickSort i Dual-pivot QuickSort. Eksperymentalnie wyznacz stałą stojącą przy czynniku $n \ln(n)$ dla liczby porównań między kluczami.

Zadanie 3. [10%]
Uzupełnij **Zadanie 1.** o algorytm hybrydowy, będący połączeniem dwóch wybranych algorytmów sortowania (z **Zadania 1.**). Zmodyfikuj algorytmy tak, by pozwalały na sortowanie danych dowolnego typu, dla których zdefiniowana jest podana przez użytkownika relacja porządku. Zaproponowany algorytm powinien skutkować lepszymi statystykami od wcześniej zaimplementowanych algorytmów sortowania. Przetestuj, czy dla danych różnego typu (podanego na etapie kompilacji lub tworzenia instancji klasy sortującej), w celu zmniejszenia czasu działania, warto jest zmieniać parametry algorytmu hybrydowego, np. moment przełączania się pomiędzy łączonymi algorytmami.