Algorytmy 2
Laboratorium: sortowanie przez kopcowanie, przez zliczanie, i kubełkowe

Przemysław Klęsk

11 listopada 2019

1 Cel

Celem zadania jest wykonanie implementacji trzech algorytmów sortujących i porównanie ich wydajności. Tymi algorytmami są:

- *sortowanie przez kopcowanie* (ang. *heapsort*),
- *sortowanie przez zliczanie* (ang. *counting sort*),
- *sortowanie kubełkowe* (ang. *bucket sort*),

Pierwszy z wymienionych jest reprezentatnem grupy algorytmów, które sortują bazując na porównaniach liczb parami, i ma złożoność obliczeniową $\Theta(n \log n)$ — asymptotycznie najlepszą możliwą w ramach tej grupy. Dwa pozostałe algorytmy nie są oparte na porównaniach i są uznawane za algorytmy liniowe co do złożoności, przy czym mają pewne ograniczenia i wymagają spełnienia pewnych założeń odnośnie wejściowych danych. Sortowanie przez zliczanie pozwala sortować tylko liczby całkowite i ma złożoność $\Theta(n+m)$, gdzie m to największa możliwa liczba w zbiorze wartości[1]. Jeżeli parametr m pozostaje ustalony lub skaluje się co najwyżej liniowo wraz z n, to efektywna złożoność obliczeniowa jest liniowa. Z kolei sortowanie kubełkowe pozwala sortować liczby rzeczywiste[2] (lub dowolne obiekty z kluczami rzeczywistymi), ale dla osiągnięcia wydajnego działania wymaga, aby liczby wejściowe były rozłożone *jednostajnie* w ustalonym przedziale. Przedział ten zostaje podzielony na pewną liczbę kubełków o równych szerokościach (np. b kubełków), realizowanych za pomocą krótkich list, do których „nadziewane" są liczby w jednym przebiegu po danych. Wynik końcowy powstaje poprzez złączenie kubełków (które muszą być uprzednio także posortowane pewnym zewnętrznym algorytmem, jeżeli zawierają więcej niż 1 element). Prowadzi to do algorytmu o złożoności obliczeniowej $\Theta(n + b)$. Powszechnie przujmuje się $b = n$ jako wybraną liczbę kubełków, ponieważ spełnienie założenia o rozkładzie jednostajnym pozwala oczekiwać,

[1] lub (w zależności od przyjętego wariantu) rozstęp zbioru wartości

[2] W implementacjach komputerowych tak naprawdę mamy do dyspozycji zmiennoprzecinkowe typy liczbowe będące podzbiorami zbioru liczb wymiernych. Niemniej, sam algorytm jako taki jest sformułowany dla liczb rzeczywistych.

że do każdego kubełka trafi średnio 1 liczba. A zatem otrzymujemy złożoność $\Theta(n+b) = \Theta(2n)$, i tym samym $\Theta(n)$.

Należy zaznaczyć, że sortowanie przez kopcowanie może być zrealizowane *w miejscu* bez nakładów pamięciowych, zaś pozostałe dwa algorytmy wymagają dodatkowej pamięci proporcjonalnej odpowiednio do m i n. Jeżeli chodzi o zewnętrzny algorytm sortujący, pomocniczy dla bucket sort, to często wykorzystuje się do tego celu sortowanie przez wstawianie (ang. *insertion sort*), ale w szczególności może to być także np. heapsort.

2 Instrukcje, wskazówki, podpowiedzi

1. Całość implementacji tego zadania nie wymaga tworzenia nowych klas.

2. Implementacja sortowania przez kopcowanie powinna wykorzystywać kopiec binarny wykonany jako własny kontener przy okazji wcześniejszego zadania laboratoryjnego (np. poprzez dołączenie odpowiedniego pliku nagłówkowego `.h`). Przy czym wymagane będą pewne rozszerzenia tej implementacji opisane w dalszej części tej instrukcji.

3. Sortowanie przez zliczanie powinno być zrealizowane jako zwykła funkcja, do której jako argumenty przekazane będą: tablica liczb całkowitych (lub ogólniej wskaźnik na taką tablicę), rozmiar tablicy, oraz liczba m określająca rozstęp zbioru wartości. Można przyjąć założenie, że dziedziną dla liczby wejściowych jest zbiór $\{0, 1, \ldots, m-1\}$. Typ zwracanego wyniku to `void` — innymi słowy funkcja ma realizować sortowanie w miejscu.

4. Należy przygotować dwa warianty (przeciążenia) funkcji realizującej sortowanie kubełkowe z rozróżnieniem typu danych wejściowej tablicy:

   - wariant przeznaczony tylko do sortowania tablic liczb całkowitych tj. `int* array`,
   - wariant wykorzystujący mechanizm szablonu (`template <typename T>`) pozwalający sortować tablice obiektów dowolnego typu tj. `T* array`.

   Wariant pierwszy tak naprawdę zawęża możliwości sortowania kubełkowego i ma być zrealizowany jako osobny przypadek tylko na potrzeby niniejszego zadania laboratoryjnego w celu możliwości przeprowadzenia porównania z sortowaniem przez zliczanie (na równych warunkach). W przypadku drugiego wariantu należy jako argumenty (oprócz tablicy oraz liczb n i m) przesłać dodatkowo wskaźniki na dwie funkcje: jedną decydującą o kluczu sortowania w ramach typu T (uwaga: klucz w ogólności zmiennoprzecinkowy), drugą stanowiącą komparator dwóch obiektów typu T (komparator będzie wykorzystany przez pomocniczy algorytm sortujący kubełki).

5. W eksperymentach porównujących wydajność algorytmów sortujących przydatne może być m.in.:

   - użycie funkcji kopiującej pamięć `memcpy(...)` w celu powielenia wejściowych tablic,
   - stworzenie funkcji pomocniczej sprawdzającej równość zawartości dwóch tablic (w celu sprawdzenia, czy wyniki sortowania są zgodne),

- losowanie liczb z szerokiego zakresu np. poleceniem: `((rand() << 15) + rand()) % m`, gdzie `m` odpowiada rozważanemu wcześniej parametrowi m,
- w razie ewentualnej potrzeby zamiany całkowitych liczb losowych na zmiennoprzecinkowe (w przypadku sortowania kubełkowego) można wykonać dzielenie np.: `rand() / (double) RAND_MAX` lub `((rand() << 15) + rand()) / (double) pow(2, 30)`.
- stworzenie funkcji zwracającej skrótową napisową reprezentację tablicy (w celu późniejszego wypisu na ekran).

3 Rozszerzenie implementacji kopca binarnego

Implementację kopca binarnego wykonaną przy okazji innego zadania laboratoryjnego należy rozszerzyć o następujące elementy:

- dodatkowy konstruktor pozwalający „wstrzyknąć" do środka kopca pewną zaalokowaną na zewnątrz tablicę (poprzez wskaźnik i podanie jej rozmiaru); konstruktor nie będzie więc alokował własnej tablicy, a powinien jedynie dokonać naprawy „wstrzykniętej" tablicy (w miejscu) w celu spełenienia warunku kopca;
- dwa warianty funkcji wykonujące naprawę kopca wg podejść top-down oraz bottom-up (konstruktor z poprzedniego punktu będzie uruchamiał jeden z nich wybrany np. za pomocą argumentu typu `bool`) — te warianty będą także podlegały sprawdzeniu wydajności w eksperymentach;
- funkcję `sort(...)` realizująca sortowanie przez kopcowanie w miejscu.

Uwaga: w zależności od przyjętego rozwiązania programistycznego, dwa ostatnie punkty powyżej mogą wymagać przekazania jako argument komparatora (wskaźnik na funkcję) lub przeciążenia operatora porównania.

4 Zawartość funkcji `main()` — dwa eksperymenty

Należy przygotować dwie wersje głównej funkcji programu (np. `main_ints()` i `main_objects()`). Będą one porównywały wydajność sortowania dla odpowiednio: liczb całkowitych (porównywane wszystkie trzy algorytmy) oraz dowolnych obiektów z pewnym kluczem zmiennoprzecinkowym (porównywane tylko heapsort i bucket sort). W obydwu eksperymentach rozmiary sortowanych tablic mają zmieniać się (kolejnymi rzędami wielkości) od 10^1 aż do 10^7. Wartość parametru m można ustalić na: 10^7 dla pierwszego eksperymentu, oraz 1.0 dla drugiego eksperymentu po uprzednim znormalizowaniu liczb losowych do przedziału $[0, 1]$ (i przechowaniu wyniku w typie `double`).

Poniższy listing pokazuje poglądowy schemat pierwszego eksperymentu:

```
int main_ints()
{
    srand(0);

    const int MAX_ORDER = 7; // maksymalny rzad wielkosci sortowanych danych
    const int m = (int) pow(10, 7); // sortowane liczby ze zbioru {0, ..., m - 1}

    for (int o = 1; o <= MAX_ORDER; o++)
    {
        const int n = (int) pow(10, o); // rozmiar tablicy z danymi

        int* array1 = new int[n];
        for (int i = 0; i < n; i++)
        {
            int rand_val = ... // tu losowanie liczby calkowitej
            array1[i] = rand_val;
        }
        ... // skrotowy wypis tablicy do posortowania (np. pewna liczba poczatkowych elementow)
        int* array2 = new int[n];
        int* array3 = new int[n];
        memcpy(array2, array1, n * sizeof(int)); // pierwsza kopia
        memcpy(array3, array1, n * sizeof(int)); // druga kopia

        // sortowanie przez zliczanie (do wykonania w miejscu)
        clock_t t1 = clock();
        counting_sort(array1, n, m);
        clock_t t2 = clock();
        ... // wypis pomiaru czasu i skrotowej postaci wynikowej tablicy

        // sortowanie przez kopcowanie (do wykonania w miejscu)
        t1 = clock();
        binary_heap<int>* bh = new binary_heap<int>(array2, n, int_cmp, true); // konstruktor kopca z
            mozliwoscia przekazania zewnetrznej tablicy (ostatni argument wskazuje kierunek naprawy:
            top-down lub bottom-up)
        bh->sort(int_cmp);
        t2 = clock();
        ... // wypis pomiaru czasu i skrotowej postaci wynikowej tablicy

        // sortowanie kubelkowe (do wykonania w miejscu)
        t1 = clock();
        bucket_sort(array3, n, m); // szczegolna wersja bucket sort tylko dla liczb calkowitych
        t2 = clock();
        ... // wypis pomiaru czasu i skrotowej postaci wynikowej tablicy

        ... // sprawdzenie zgodnosci tablic array1, array2, array3 i wypis informacji o zgodnosci na
            ekran

        delete[] array1, array2, array3;
    }
}
```

Poniższy listing pokazuje poglądowy schemat drugiego eksperymentu:

```
int main_some_objects()
{
    const int MAX_ORDER = 7; // maksymalny rzad wielkosci sortowanych danych
    const double m_double = (double) pow(2, 30); // mianownik przy ustalaniu losowej liczby
        zmiennoprzecinkowej

    for (int o = 1; o <= MAX_ORDER; o++)
    {
        const int n = (int) pow(10, o); // rozmiar tablicy z danymi

        some_object** array1 = new some_object*[n];
        for (int i = 0; i < n; i++)
        {
            some_object* so = new some_object();
            so->field_1 = ((rand() << 15) + rand()) / m_double; // przykladowy sposob wylosowania pola
                typu double (ktore bedzie stanowilo klucz sortowania)
            so->field_2 = 'a' + rand() % 26; // przykladowy sposob wylosowania pola typu char
            array1[i] = so;
        }
        ... // skrotowy wypis tablicy do posortowania (np. pewna liczba poczatkowych elementow)
        some_object** array2 = new some_object*[n];
        memcpy(array2, array1, n * sizeof(some_object*)); // kopia

        // sortowanie przez kopcowanie
        clock_t t1 = clock();
        binary_heap<some_object*>* bh = new binary_heap<some_object*>(array1, n, some_objects_cmp,
            true); // konstruktor kopca z mozliwoscia przekazania zewnetrznej tablicy (ostatni
            argument wskazuje kierunek naprawy: top-down lub bottom-up)
        bh->sort(some_objects_cmp);
        t2 = clock();
        ... // wypis pomiaru czasu i skrotowej postaci wynikowej tablicy

        // sortowanie kubelkowe
        t1 = clock();
        bucket_sort<some_object*>(array2, n, 1.0, some_object_key_double, some_objects_cmp); // trzeci
            argument wskazuje, ze liczby sa z przedzialu [0, 1]
        t2 = clock();
        ... // wypis pomiaru czasu i skrotowej postaci wynikowej tablicy

        ... // sprawdzenie zgodnosci tablic array1, array2 i wypis informacji o zgodnosci na ekran

        delete[] array1, array2;
    }
}
```

5 Sprawdzenie antyplagiatowe — przygotowanie wiadomości e-mail do wysłania

1. Kod źródłowy programu po sprawdzeniu przez prowadzącego zajęcia laboratoryjne musi zostać przesłany na adres *algo2@zut.edu.pl*.

2. Plik z kodem źródłowym musi mieć nazwę wg schematu: `nr_albumu.algo2.nr_lab.main.c` (plik

może mieć rozszerzenie `.c` lub `.cpp`). Przykład: `123456.algo2.lab06.main.c` (szóste zadanie laboratoryjne studenta o numerze albumu 123456). Jeżeli kod źródłowy programu składa się z wielu plików, to należy stworzyć jeden plik, umieszczając w nim kody wszystkich plików składowych.

3. Plik musi zostać wysłany z poczty ZUT (zut.edu.pl).

4. Temat maila musi mieć postać: `ALGO2 IS1 XXXY LAB06`, gdzie `XXXY` to numer grupy (np. `ALGO2 IS1 210C LAB06`).

5. W pierwszych trzech liniach pliku z kodem źródłowym w komentarzach muszą znaleźć się:

   - informacja identyczna z zamieszczoną w temacie maila (linia 1),
   - imię i nazwisko autora (linia 2),
   - adres e-mail (linia 3).

6. Mail nie może zawierać żadnej treści (tylko załącznik).

7. W razie wykrycia plagiatu, wszytkie uwikłane osoby otrzymają za dane zadanie ocenę 0 punktów (co jest gorsze niż ocena 2 w skali $\{2, 3, 3.5, 4, 4.5, 5\}$).