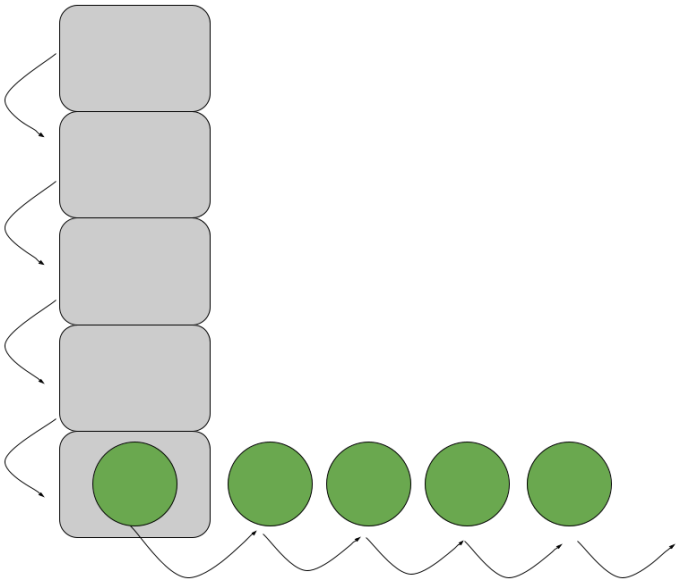
Miłosz Klim
Technologie Komputerowe, III semestr
6 II 2023 r.

Wyszukiwanie najmniejszego i największego elementu ciągu {ai} zapisanego w tablicy z haszowaniem, jeżeli 0.99 ≤ ai ≤999.99

Wstęp

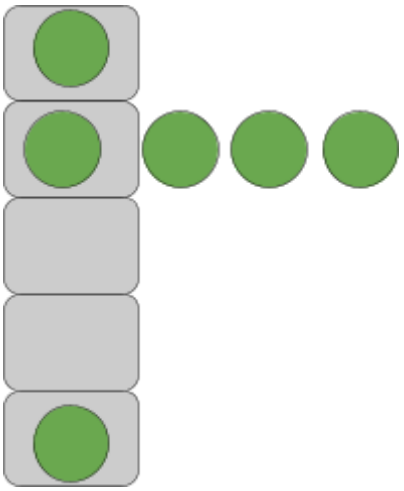
Tablice mieszające są jednym ze sposobów implementacji tablic asocjacyjnych, przechowujących dane w sposób umożliwiający możliwie najszybszy dostęp. Z tego powodu są one chętnie stosowane w kompilatorach i interpreterach języków dynamicznych (Python, JS, Lua), bazach danych, tablicach routingu a nawet w systemach cachowania danych.
Z technicznego punktu widzenia są one do pewnego stopnia zwykłymi tablicami, na których jednak zamiast działać używając bezpośrednio indeksów, odnosi się za pomocą funkcji haszującej przekształcających zadany klucz w indeks. Umożliwia to niezwykle efektywne wyszukiwanie konkretnych elementów, ograniczając obszar przeszukiwania dokładnie do komórki wyznaczonej przez funkcję.
Skutkiem ubocznym działania funkcji haszujących jest zjawisko kolizji, zachodzące w momencie gdy dwa różne elementy są przypisywane do tego samego miejsca w tablicy. W swoim algorytmie zastosowałem metodę łańcuchową z użyciem list jednokierunkowych, bez kontroli powtarzających się elementów. Kolejne elementy są dodawane na koniec listy. W celu wyszukania najmniejszego lub największego wyrazu algorytm musi przeszukać całą listę, co oznacza wykonanie n/m operacji porównania, gdzie n oznacza ilość przechowywanych elementów (np. wyrazów ciągu), a m ilość komórek tworzących tablicę.
W praktyce zdarzają się pewne zbiory danych, do których możliwe jest przygotowanie tzw. doskonałej funkcji mieszającej, która to nigdy nie powoduje kolizji. Istnieją specjalne algorytmy umożliwiające wyznaczanie takich funkcji np. algorytmy Cichelliego i FHCD dla wyrazów i zdań, oraz CHD dla dowolnych danych.
Przy wyznaczaniu ostatecznej złożoności algorytmu należy pamiętać, że cały ciąg zostaje rozdzielony na wiele mniejszych list. Poza porównaniami kolejnych wyrazów listy, znaczące są również porównania pierwszych wyrazów list w kolejnych komórkach. Pseudolosowa natura ciągu nie daje pewności, że w danej komórce znajdzie się cokolwiek. Dlatego należy najpierw pobrać element znajdujący się pod adresem sprawdzanej komórki i przyrównać go do wartości nullptr, oznaczającej brak wartości.
W omawianym algorytmie, operacją dominującą jest porównanie. Liczone są zarówno porównania dwóch liczb, jak i porównanie (sprawdzenie) wartości nullptr.
Teoretyczna złożoność algorytmu dla ciągu o długości n i tablicy o m komórkach prezentuje się następująco:

Wariant pesymistyczny:
W tym przypadku wszystkie elementy zostaną umieszczone w najbardziej skrajnej do kierunku przeszukiwania komórce (pierwszej dla poszukiwań elementu największego i ostatniej dla najmniejszego), co wymaga $n+m$ operacji porównań.



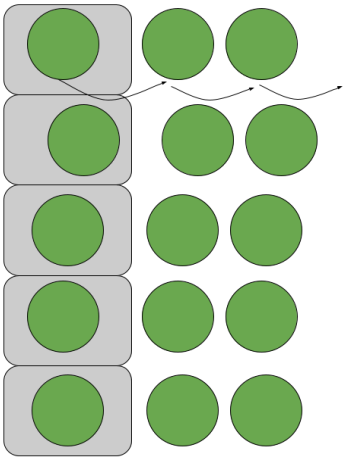
***Rys. 1**. Przykład wariantu pesymistycznego dla wyszukiwania najmniejszego wyrazu ciągu - wszystkie elementy w najmniej korzystnej komórce. Ostatnie porównanie jest sprawdzeniem końca listy.*

Wariant optymistyczny:
W tym przypadku element najmniejszy lub największy zajmują samotnie całą objętość najkorzystniejszej komórki, bez powtórzeń, co umożliwia znalezienie elementu używając dwóch porównań. Pierwsze sprawdzające czy w dana komórka nie jest pusta, i drugie sprawdzające czy istnieje następnik.
Komórkami najkorzystniejszymi są kolejno pierwsza dla elementu najmniejszego i ostatnia dla elementu największego.



***Rys. 2.** Przykład wariantu optymistycznego dla wyszukiwania najmniejszego wyrazu ciągu - element znajduje się samotnie w najkorzystniejszej komórce*

Wariant oczekiwany (przeciętny):
W tym przypadku wszystkie elementy ciągu są równomiernie rozłożone w całej objętości tablicy, co wymaga *n/m* porównań



Rys. 3. *Przykład wariantu oczekiwanego - wszystkie elementy ciągu są równomiernie rozłożone w całej objętości tablicy. Ostatnie porównanie jest sprawdzeniem końca listy.*

Celem projektu jest zaimplementowanie zadanego algorytmu i analiza jego złożoności obliczeniowej.

Metodyka

Program napisałem w języku C++, używając środowiska Visual Studio Code, kompilatora MSVC i z myślą o uruchamianiu go w systemach Windows 10 i 11. Kod źródłowy jest dostępny w publicznym repozytorium na platformie GitHub[1].

Podstawą struktury danych jest tablica wskaźników. Każde jej pole domyślnie przyjmuje wartość nullptr, oznaczając w ten sposób, że jest puste.
Chcąc wstawić kolejny element ciągu do tablicy, poddaję go procesowi haszowania, używając do tego mojej autorskiej funkcji:

[1] https://github.com/DAXPL/Algorithms-and-Data-Structures-Project

```
//Funkcja haszująca podany klucz, dla zadanego rozmiaru tablicy
int hashKey(int key, int size)
{
   int hashedKey = (key-minRandom)/((maxRandom-minRandom)/size);
   if(hashedKey>=size)
   {
      hashedKey=size-1;
   }

   return hashedKey;
}
```

Znając górną i dolną granicę wartości jakie może przyjąć każdy wyraz ciągu, oraz znając liczbę komórek tablicy (zmienna *size*), jestem w stanie każdemu wyrazowi ciągu, przyznać odpowiednie miejsce. Przyjęta formuła zapewnia, że w pierwszej komórce znajdą się wszystkie wyrazy mniejsze od *1/size*, w drugiej większe lub równe *1/size* i mniejsze od *2/size* itd. Problem stanowi klucz 99999, który po hashowaniu powinien zostać umieszczony w komórce *size*, co nie jest możliwe. W języku C++ tablice mają zakres od 0 do *size-1*, dlatego ten konkretny klucz zostanie umieszczony w komórce *size-1*.
Tablica wypełniona wyrazami zgodnie z metodą *hashKey()*, posiada najmniejszy element w pierwszej niepustej komórce, zaś największy w ostatniej niepustej.
Pewnym charakterystycznym dla tablic z haszowaniem problemem są kolizje, powstające w momencie gdy dwa różne elementy zostają przypisane do tej samej komórki. Zachodzi wtedy potrzeba wprowadzenia kolejnej struktury przechowującej te elementy.
Proponowanym przeze mnie rozwiązaniem problemu kolizji jest porcjowanie (nazywane również w literaturze metodą łańcuchową), przy pomocy list jednokierunkowych składających się z kolejnych instancji struktury *Ai*. Wspomniana struktura zawiera zmienną przechowującą adres pamięci, pod którym znajduje się jej następnik. Metoda ta znacząco ułatwia usuwanie i wstawianie kolejnych elementów, kosztem długiego czasu wyszukiwania dla nich miejsca. Operacje modyfikujące jej zawartość muszą zostać poprzedzone sekwencyjnym przeszukaniem całej struktury listy, w celu znalezienia adresu pamięci poprzednika, dlatego w wielu praktycznych implementacjach tablic haszujących używa się dodatkowych pól w pamięci, przechowujących adres ostatniego elementu na liście.
Wartość *nullptr* pola *next* oznacza, że algorytm natrafił na ostatni element w liście.
Wartym uwagi jest pole *number* typu całkowitego oraz metoda *GetDoubleValue()* typu double. Ze względu na duże ograniczenia związane ze sposobem w jaki komputery działają na liczbach zmiennoprzecinkowych, zdecydowałem się na działanie na liczbach całkowitych i korygowanie wyświetlanych wyników.
Taka modyfikacja znacząco ułatwia i przyspiesza działanie programu, a jednocześnie nie ma wpływu na złożoność algorytmu, tak długo jak konsekwentnie trzymam się przyjętego założenia

```
//Struktura wyrazu ciągu
struct Ai
{
   Ai* next;//następnik
   int number;//wartość własna
   double GetDoubleValue()
   {
      int dd = number;
      return dd/100;
   }
};
```

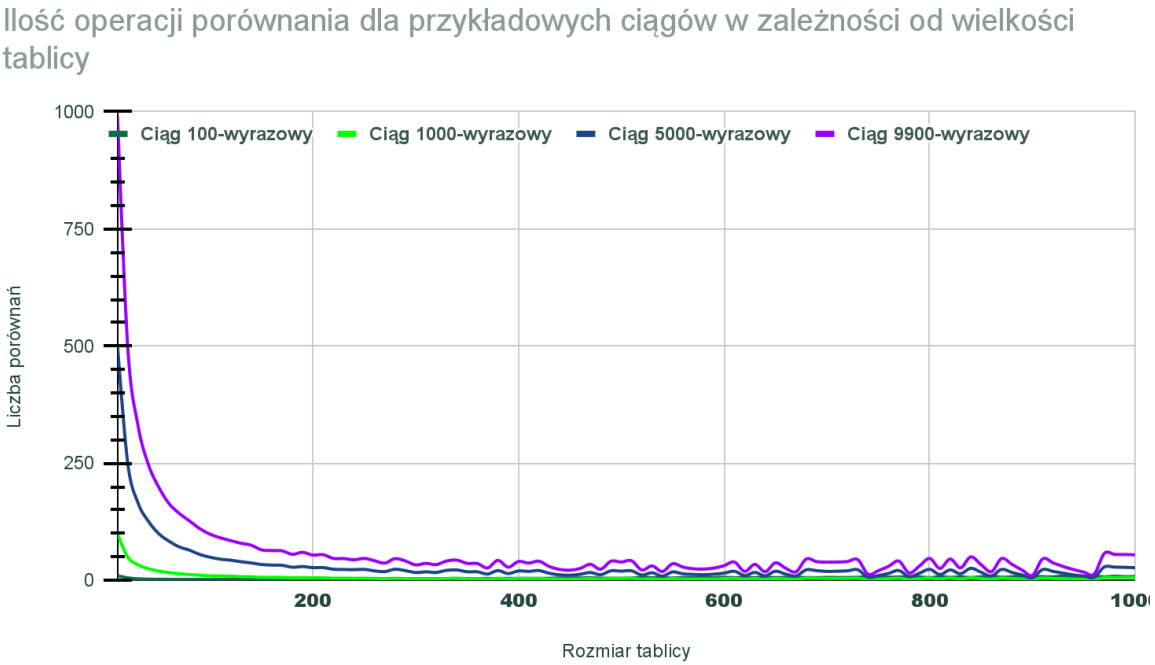
Klucze są generowane w sposób pseudolosowy, przy użyciu algorytmu **Mersenne Twister**[2], z użyciem poniższych ustawień. Zakres jest wyznaczany przez stałe *minRandom* i *maxRandom*, równe kolejno 99 i 99999, co umożliwia generowanie liczb w zakresie wymaganym przez temat projektu.

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<unsigned int> dis(minRandom, maxRandom);
```

Wyniki projektu

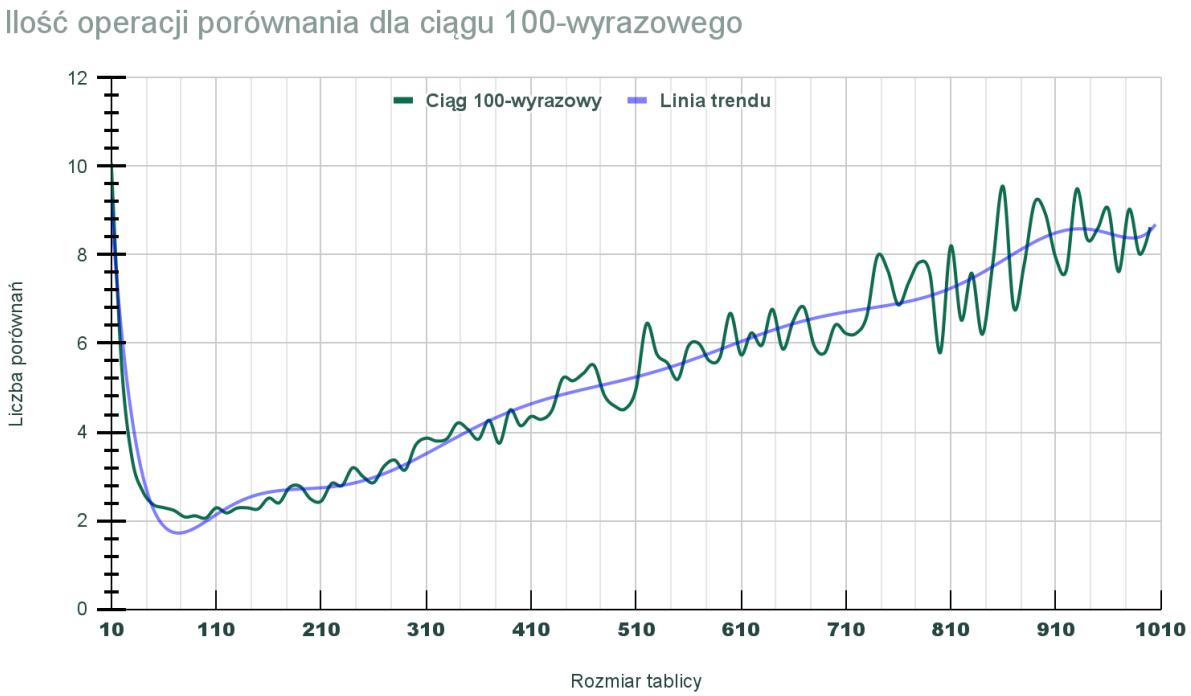
Chcąc zbadać zachowanie algorytmu zarówno dla zmiennej długości tablicy, jak i zmiennej długości ciągu, w centrum programu umieściłem trzy zagnieżdżone pętle for, umożliwiające wykonanie zadanej liczby pomiarów uśredniających dla każdej kombinacji zdefiniowanych rozmiarów tablic i ciągów. Każdy ciąg składał się z wyrazów pseudolosowych, wygenerowanych przy pomocy algorytmu Mersenne Twister, w sposób opisany w dziale "Metodyka".
Wynikiem działania programu był plik dane.csv, w którym znalazło się 10 tysięcy wierszy zawierających pomiary złożoności obliczeniowej dla różnych kombinacji długości tabeli oraz ciągu. Każdy wiersz powstał w wyniku uśrednienia stu innych pomiarów, co oznacza, że swoje rozważania będę opierał na podstawie analizy miliona różnych kombinacji.
Arkusz ze wszystkimi danymi zorganizowanymi w tablice przestawne jest dostępny w usłudze arkusze google[3].

[2]https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine
[3]https://docs.google.com/spreadsheets/d/1pn1ZBARtU50h_uiB4q4HI5zPUk0zBuFkXefsDagFy8g/edit?usp=sharing



Wyk. 1. *Wykres przedstawiający zależność między liczbą operacji charakterystycznych, a rozmiarem tablicy dla wybranych ciągów o różnej długości*

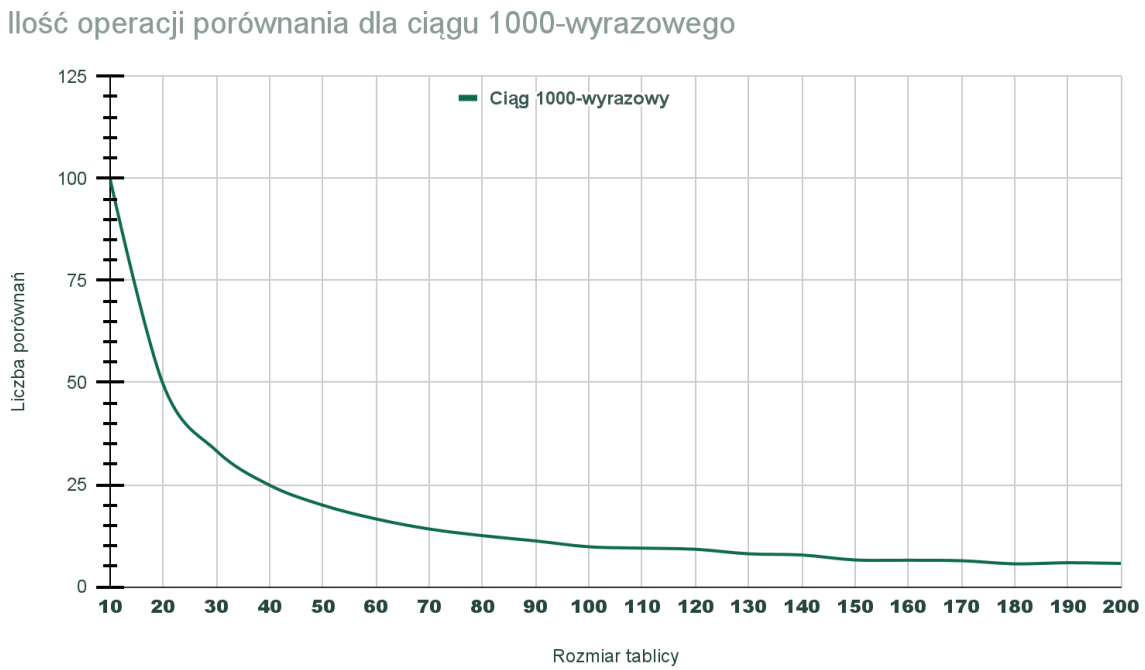
Bardzo ważną kwestią wartą poruszenia przy omawianiu zachowania tablic z haszowaniem jest zależność między długością ciągu a wielkością tablicy. Współczynnik wypełnienia określa stosunek ilości komórek przechowujących dane do całkowitej długości tablicy. Zasadniczo większa tablica pozwala podzielić ciąg na mniejsze podproblemy, ale jeśli jej wielkość będzie nieproporcjonalnie duża do długości ciągu (niski współczynnik wypełnienia), to algorytm będzie zmuszony przeszukiwać dużą liczbę pustych komórek. Doskonale to widać na przykładzie stu-elementowego ciągu:



Wyk. 2. *Liczba operacji porównania dla ciągu 100-wyrazowego, wraz z linią trendu*

Początkowo zwiększanie rozmiaru tablicy powoduje zmniejszanie liczby porównań, ale w pewnym momencie komórek pustych jest znacznie więcej niż przechowywanych wyrazów, co wydłuża proces przeszukiwania.

Niedobrze jest również kierować w stronę drugiej skrajności, czyli przypadku, gdzie wielkość tablicy jest znacznie mniejsza od długości ciągu. W takiej sytuacji co prawda algorytm nie traci dużo czasu na szukanie niepustej komórki, ale wciąż jest ograniczony przeszukiwaniem sekwencyjnym dużej listy jednokierunkowej umieszczonej w tej komórce. Używając klasycznego rachunku prawdopodobieństwa można oszacować długość tej listy na *n/m*, co jest bardzo niekorzystne dla właśnie dla dużych n (długość ciągu) i małych m (wielkość tablicy).



Wyk. 3. *Liczba operacji porównania dla ciągu 1000-wyrazowego, w zależności od rozmiaru tablicy z haszowaniem*

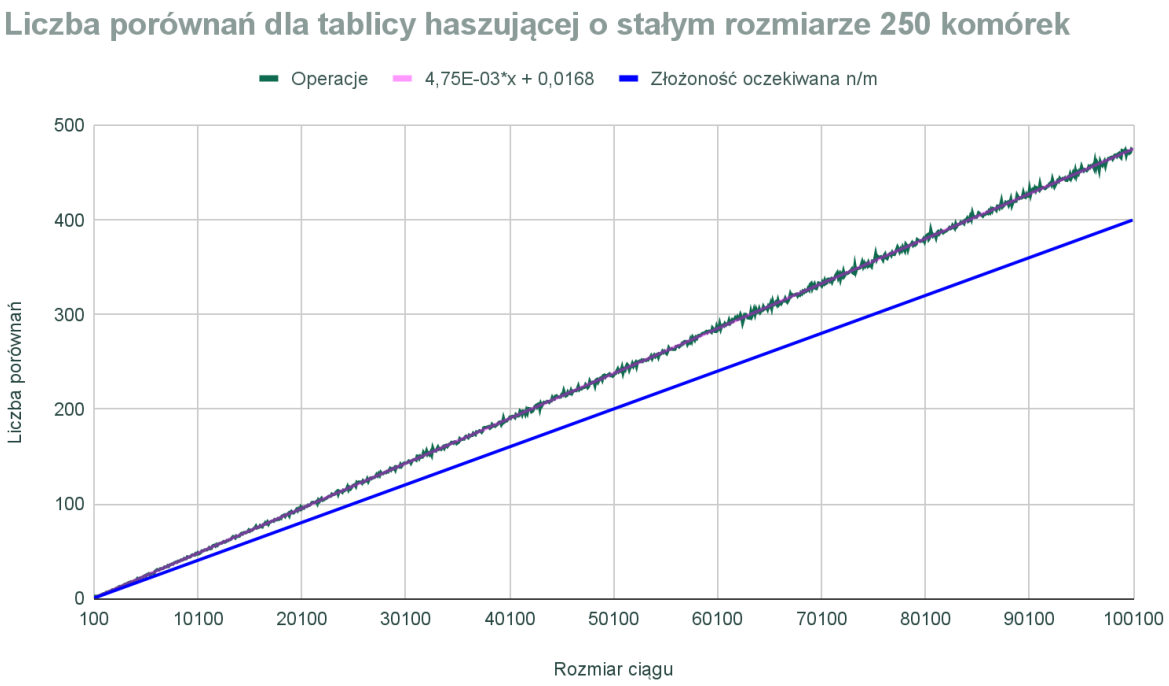
Powyższy przykład zachowania ciągu 1000-wyrazowego dla małych tablic dobrze obrazuje opisywane zjawisko. Ilość operacji dla m=10 jest dużo niższa niż dla zwykłego przeszukiwania sekwencyjnego, ale wciąż jest to wydajność daleka od tej oczekiwanej w przypadku tablic z haszowaniem.

Wyznaczona przez program linia trendu (***Wyk. 2)***, będąca wielomianem dziesiątego stopnia, osiąga minimum globalne dla x w okolicach 74.8601, co zostało potwierdzone kalkulatorem ekstremów wolframalpha

$$\min\{13.2 - 0.484\,x + 0.00791\,x^2 - 6.62\times10^{-5}\,x^3 + 3.24\times10^{-7}\,x^4 - 9.9\times10^{-10}\,x^5 + 1.93\times10^{-12}\,x^6 - 2.42\times10^{-15}\,x^7 + 1.88\times10^{-18}\,x^8 - 8.21\times10^{-22}\,x^9 + 1.54\times10^{-25}\,x^{10}\} \approx 1.68088 \text{ at } x \approx 74.8601$$

Rys. 4. *Wynik działania kalkulatora wolframalpha (https://www.wolframalpha.com)*

Tablice z haszowaniem powinny być stosowane w sytuacjach, w których rozmiar ciągu jest duży w stosunku do rozmiaru tablicy, w której ma zostać umieszczony. W praktyce, rozwiązania bazujące na omawianej strukturze danych starają się utrzymywać współczynnik wypełnienia w okolicach 0,7-0,8.



Wyk. 4. *Liczba operacji porównania dla różnych długości ciągu, oraz stałego rozmiaru tablicy haszującej (250 komórek)*

Do analizy złożoności algorytmicznej dla stałego rozmiaru tablicy (250 komórek) i zmiennej długości ciągu użyte zostały dane powstałe w wyniku uśrednienia 50 pomiarów i obejmowały one długości ciągu od 100 do 100 000 wyrazów, z krokiem 100. Cała reszta ustawień pozostała niezmienna w stosunku do poprzedniej serii pomiarów, omawianej we wcześniejszej części raportu.

Wyznaczona złożoność rzeczywista, wyrażana wzorem ***4,75e(-3)*n + 0,0168*** nie przekracza złożoności pesymistycznej ($n+m$), ani nie przyjmuje wartości poniżej złożoności optymistycznej. Odchylenie od złożoności oczekiwanej (n/m) wynika z pseudolosowej natury ciągu.
Rzeczywista złożoność algorytmu wyszukującego najmniejszy lub największy element ciągu $\{ai\}$ zapisanego w tablicy z haszowaniem, jeżeli $0.99 \leq ai \leq 999.99$, w mojej implementacji jest złożonością liniową, zapisywaną jako ***O(n)***.

Wnioski

- Wyznaczona liniowa złożoność algorytmu jest zgodna z ogólnie przyjętą teorią[4]. Odchylenia związane są z pseudolosową naturą ciągu i koniecznością wyszukania pierwszej niepustej komórki.
- Omawiany algorytm działa najefektywniej dla współczynnika wypełnienia w okolicach 0,75.

Źródła i literatura

1. https://eduinf.waw.pl/inf/alg/001_search/0099a.php
2. https://eduinf.waw.pl/inf/alg/001_search/0067e.php
3. https://www.geeksforgeeks.org/hashing-data-structure/
4. https://home.math.uni.lodz.pl/horzel/wp-content/uploads/sites/3/2017/04/haszowanie.pdf
5. Banachowski Lech - Algorytmy i struktury danych
6. https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine

[4] Według eduinf.waw.pl (Źródła i literatura, podpunkt 1)