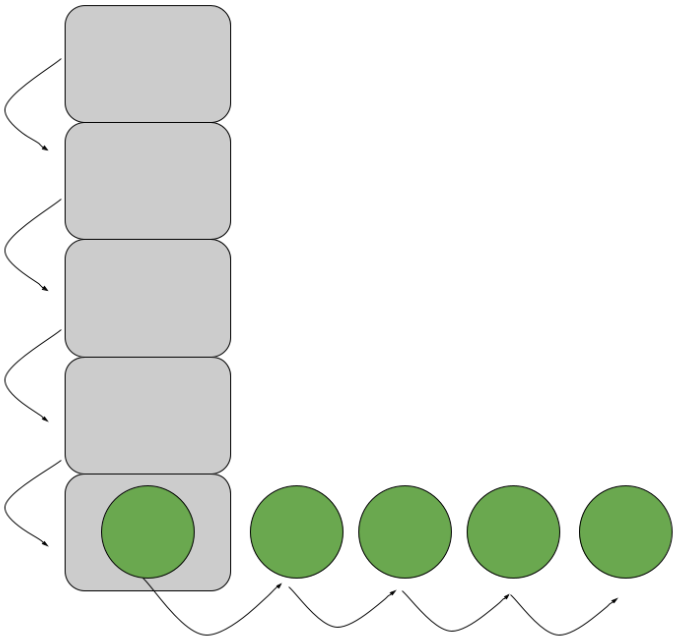
Wstęp

Zadaniem programu jest wyszukanie najmniejszego i największego elementu ciągu $\{a_i\}$ zapisanego w tablicy z haszowaniem, jeżeli $0.99 \leq a_i \leq 999.99$.

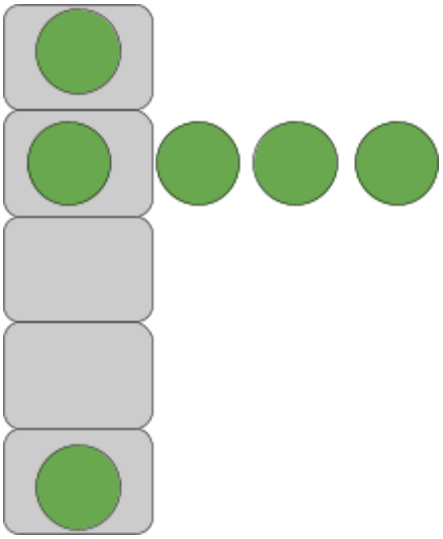
Tablice mieszające są jednym ze sposobów implementacji tablic asocjacyjnych, przechowujących dane w sposób umożliwiający możliwie najszybszy dostęp. Z technicznego punktu widzenia są zwykłymi tablicami, w których jednak zamiast działać bezpośrednio na indeksach, odnosi się do nich za pomocą funkcji haszującej przekształcających zadany klucz w indeks. Umożliwia to niezwykle efektywne wyszukiwanie konkretnych elementów, ograniczając obszar przeszukiwania 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. W celu wyszukania najmniejszego lub największego wyrazu algorytm musi przeszukać całą listę, co oznacza wykonanie n/m operacji, a więc złożoność wzrasta liniowo wraz ze wzrostem ilości 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. Losowa 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.
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 $n+m-2$ operacji porównań.



***Rys. 1**. Przykład wariantu pesymistycznego dla wyszukiwania najmniejszego wyrazu ciągu - wszystkie elementy w najmniej korzystnej komórce*

Wariant optymistyczny:
W tym przypadku element najmniejszy lub największy zajmują samotnie całą objętość najkorzystniejszej komórki, bez powtórzeń, co umożliwia znalezienie elementu używając zaledwie jednego porównania.
Komórkami najkorzystniejszymi są kolejno pierwsza (indeks równy 0) dla elementu najmniejszego i ostatnia dla elementu największego.



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

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 repozytorium na platformie GitHub.
https://github.com/DAXPL/Algorithms-and-Data-Structures-Project

Podstawą struktury danych jest tablica wskaźników. Każde jej pole domyślnie przyjmuje wartość nullptr, oznaczając w ten sposób, że jest puste.

```
for(int z=1;z<=tables;z++)//pomiary dla różnych rozmiarów tablicy haszującej
    {
        tableSize = 10*z;
        numbers = new Ai*[tableSize];
        PrepareTable();
```

```
//Przygotowanie obszaru w pamięci dla tablicy
void PrepareTable()
{
    for(int i=0;i<tableSize;i++)
    {
        numbers[i]=nullptr;
    }
}
```

Chcąc wstawić kolejny element ciągu do tablicy, poddaję go procesowi haszowania, używając do tego mojej autorskiej funkcji:

```
//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 ilość 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* wartości górnej, 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.
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;
    }
};
```

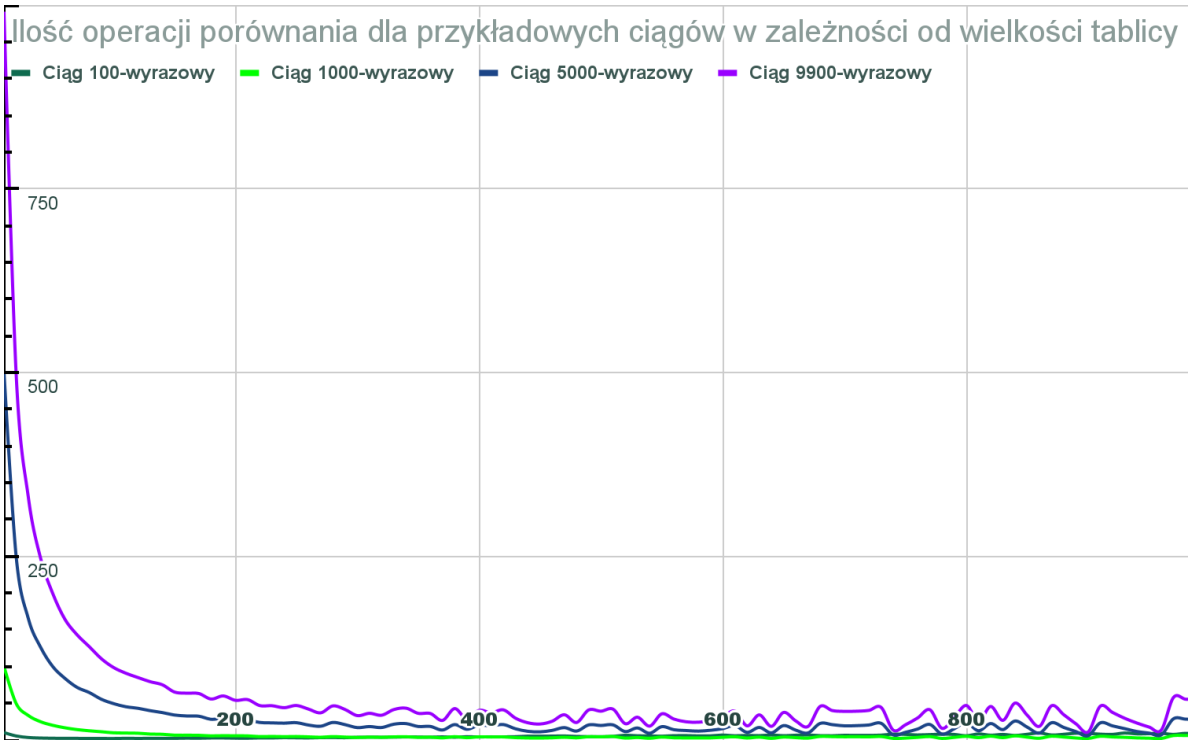
Wyniki projektu

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 ilości pomiarów uśredniających dla każdej kombinacji zdefiniowanych rozmiarów tablic i ciągów:

```
for(int z=1;z<=tables;z++)//pomiary dla różnych rozmiarów tablicy
{
    tableSize = 10*z;
    numbers = new Ai*[tableSize];
    PrepareTable();
   for(int i=10;i<=maxNumers;i*=10)//pomiary dla różnych długości ciągu
   {
      double compSmMed =0;//Średnia porównań dla najmniejszego elementu
      double compLgMed =0;//Średnia porównań dla największego elementu
      double compMed =0;
      for(int j=0;j<measurements;j++)//powtórzenia dla uśrednienia
      {
         //porównania dla tej serii pomiarów
         int compSm =0;
         int compLg =0;

         //Wykonanie serii pomiarów
         SeriesOfMeasurements(outputFile,i,compSm,compLg);

         //Sumowanie liczby operacji
         compSmMed+=compSm;
         compLgMed+=compLg;
      }

      //wyznaczanie średniej liczby operacji
      compMed += compSmMed + compLgMed;
      compSmMed/=measurements;
      compLgMed/=measurements;
      compMed/=2*measurements;
   }
}
```

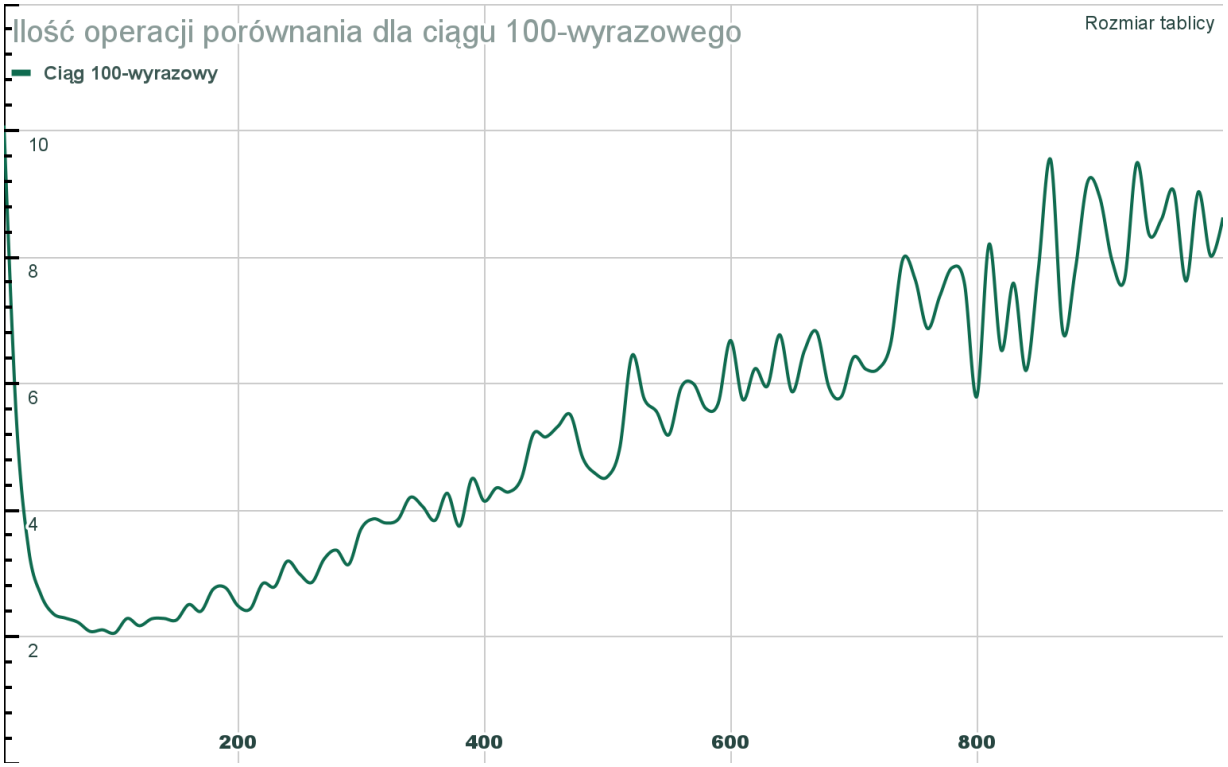
Wynikiem działania programu był plik dane.csv, w którym znalazło się 10 tysięcy wierszy zawierających pomiary złożoności obliczeniowej dla różnych kombinacji długości tabeli oraz ciągu. Każdy wiersz powstał w wyniku uśrednienia stu innych pomiarów, co oznacza, że swoje rozważania będę opierał na podstawie analizy miliona różnych kombinacji.

Arkusz ze wszystkimi danymi zorganizowanymi w tablicę przestawną jest dostępny w usłudze arkusze google.
https://docs.google.com/spreadsheets/d/1pn1ZBARtU50h_uiB4q4HI5zPUk0zBuFkXefsDagFy8g/edit?usp=sharing



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

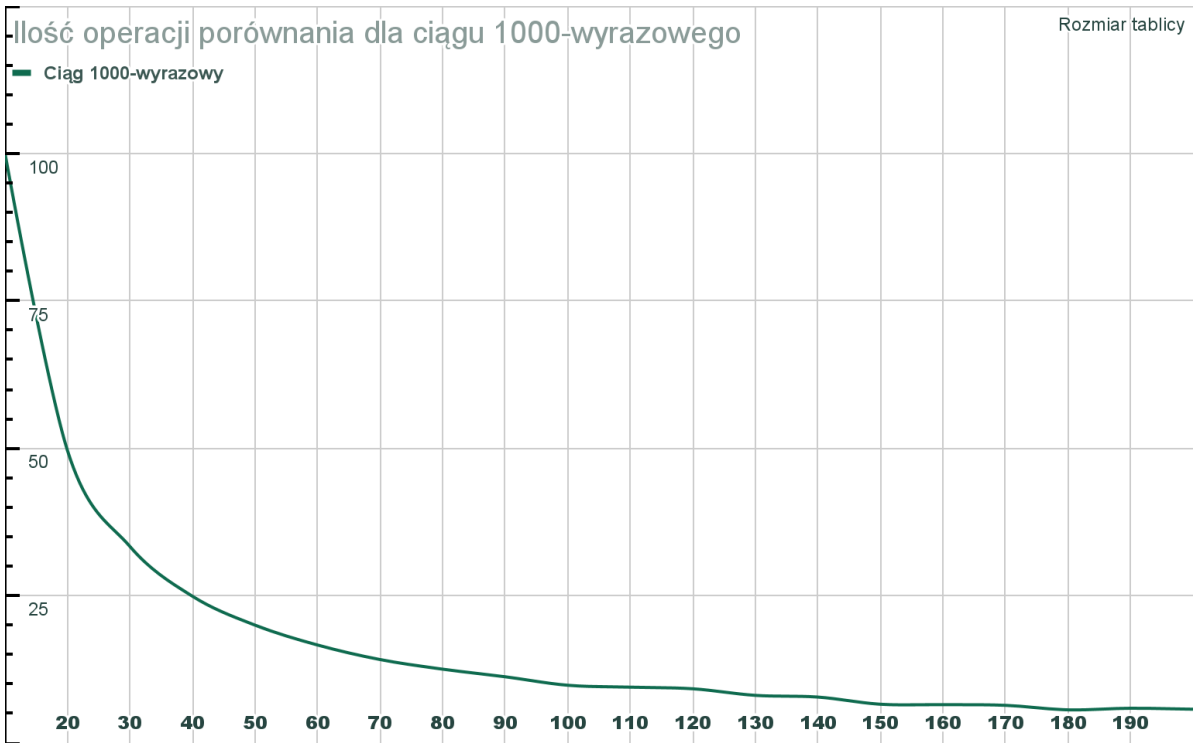
Bardzo ważną kwestią wartą poruszenia przy omawianiu zachowania tablic z haszowaniem jest zależność między długością ciągu (uniwersum kluczy) a wielkością tablicy, nazywana współczynnikiem wypełnienia. Zasadniczo większa tablica pozwala podzielić ciąg na mniejsze podproblemy, ale jeśli jej wielkość będzie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



Wyk. 2. *Ilość operacji porównania dla ciągu 100-wyrazowego*

Początkowo zwiększanie rozmiaru tablicy powoduje zmniejszanie ilości 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 dużych n (długość ciągu) i małych m (wielkość tablicy).

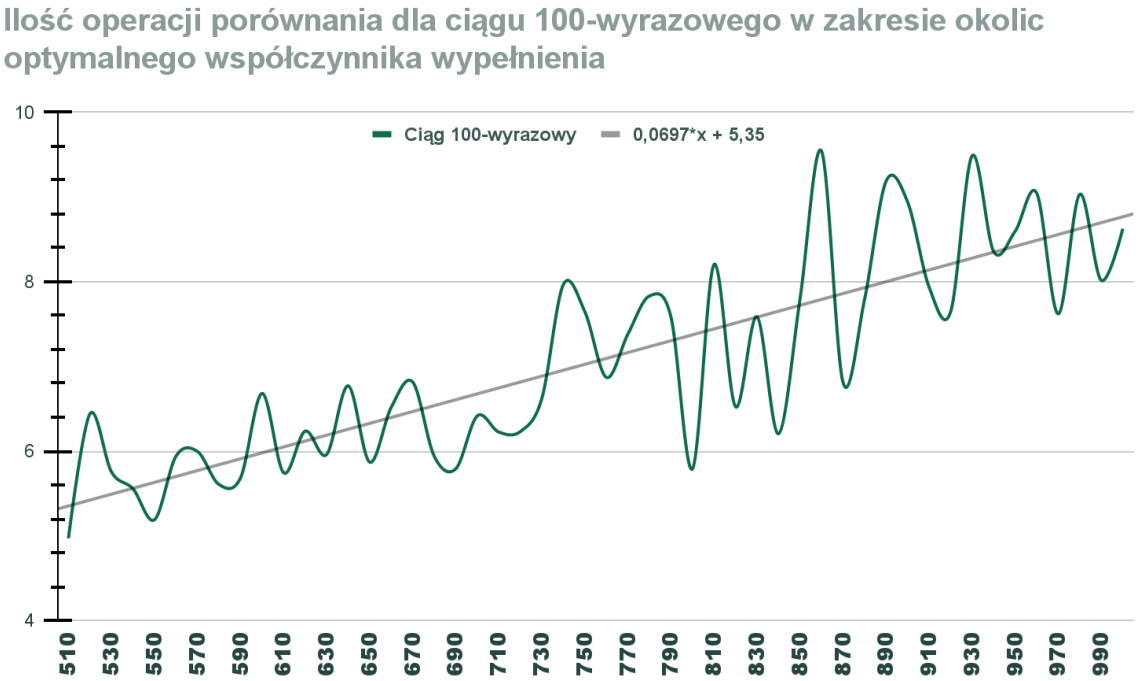


Wyk. 3. *Ilość operacji porównania dla ciągu 1000-wyrazowego*

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.

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

Poruszając się w obrębie wielkości zalecanych dla omawianej struktury danych, złożoność obliczeniowa zgadza się ze złożonością pesymistyczną O(n). Wyznaczony wzór złożoności oczekiwanej jest opisywany wzorem 0,0697*x + 5,35.



Wyk. 4 .*Ilość operacji porównania dla ciągu 100-wyrazowego w zakresie okolic optymalnego współczynnika wypełnienia*

Własność ta znajduje swoje odzwierciedlenie w praktycznych zastosowaniach tablicy z haszowaniem. Jest ona bowiem szeroko stosowana w bazach danych, gdzie czas dostępu do bardzo konkretnych danych jest kluczowy.

Wnioski

- Wyznaczona złożoność oczekiwana O(n) algorytmu jest zgodna z ogólnie przyjętą teorią. Odchylenia związane są z losową naturą ciągu i koniecznością wyszukania pierwszej niepustej komórki.
- Omawiany algorytm działa efektywnie dla współczynnika wypełnienia w okolicach 0,7 .
- Pewną możliwą optymalizacją programu byłoby wyznaczanie i przechowywanie adresów skrajnych niepustych komórek jeszcze na etapie operacji wstawiania. Pozwoliłoby to wyeliminować problem zbyt dużych m do wartości n kosztem większego zużycia pamięci (adresy musiałyby być przechowywane poza strukturą danych) oraz wydłużeniem operacji wstawiania.

Źródła

- https://eduinf.waw.pl/inf/alg/001_search/0099a.php
- https://eduinf.waw.pl/inf/alg/001_search/0067e.php
- https://www.geeksforgeeks.org/hashing-data-structure/