Ezoteryczne Kartki
Algorytmy randomizowane II

Jakub Bachurski

wersja 1.0.1.1

Gdy nie wiesz, co zrobić, zrób coś losowego.

Anonim

1 Hashmapy

Bardzo często używanymi w informatyce strukturami są wszelkiego rodzaju hashmapy. Są najczęściej stosowane jako struktury działające jako zbiory lub słowniki (struktury przypisujące do jednego typu wartości – „kluczy" – drugi typ – „wartości"). W C++ są dostępne pod nazwami `std::unordered_set` oraz `std::unordered_map`.

1.1 Idea

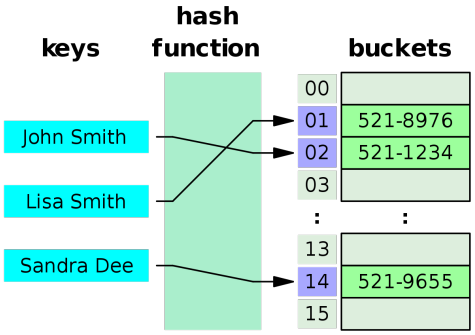
Hashmapy działają na bardzo prostej zasadzie: opiszmy funkcję hashującą

$$\mathcal{H} : K \to \mathbb{Z}_n$$

Gdzie K to zbiór możliwych kluczy, a $\mathbb{Z}_n$ oznacza zbiór $\{0, 1, 2, ..., n-1\}$.

Następnie stwórzmy sobie wiaderka (*buckets* albo *slots*), będące komórkami tablicy wielkości n. Klucz k trafi do $\mathcal{H}(k)$-tego z nich. Przy odrobinie szczęścia, wartości $\mathcal{H}$ nie będą się powtarzać i będziemy w spokoju operować na hashmapie. Podsumujmy krótko możliwe operacje (optymistycznie w $O(1)$):

- `insert(k)` oraz `erase(k)` przeprowadzamy odpowiednio modyfikując wiaderko $\mathcal{H}(k)$.
- `find(k)` wykonujemy sprawdzając, czy wiaderko $\mathcal{H}(k)$ zawiera poszukiwany element.
- Iterację po wszystkich elementach możemy wykonać przechodząc po wszystkich wiaderkach, w czasie $O(n)$. Warto zauważyć, że kolejność iteracji jest nieznana.



Na obrazku powyżej przedstawione jest przypisywanie kluczy do odpowiednich wiaderek[1].

W tym prostym pomyśle jest jeden główny problem: co jeżeli natrafimy na parę kluczy (k_1, k_2), dla których $\mathcal{H}(k_1) = \mathcal{H}(k_2)$? Taką sytuację nazywamy kolizją hashy i jest ona w oczywisty sposób niepożądana. Zatem chcemy, by $\mathcal{H}$ dawało możliwie losowe wartości, zmniejszając w ten sposób prawdopodobieństwo przypadkowej kolizji.

Oczywiście, algorytm hashujący obliczający $\mathcal{H}$ musi działać deterministycznie, by wielokrotne hashowanie tych samych kluczy dawało ten sam indeks wiaderka. Poza tym, najlepiej, żeby trudno było skonstruować przypadki pesymistyczne algorytmu (duży zbiór kluczy kolidujących ze sobą), oraz żeby działał on jak najszybciej, bo będziemy często przeliczać wartości $\mathcal{H}$. Dlatego często wykorzystuje się arytmetykę modularną (która, odpowiednio wykorzystana, może dawać łatwe do obliczenia pseudolosowe wartości) oraz operacje bitowe (które są szybkie i ogólnodostępne).

Nietrudno zauważyć, że kolizji właściwie nie da się uniknąć: jeżeli $n < |K|$ muszą istnieć kolizje (a jeżeli wybralibyśmy $n \geqslant |K|$ to niepotrzebnie wprowadzamy hashe). Musimy zatem rozpatrzyć sposoby ich rozwiązywania.

1.2 Rozwiązywanie kolizji

A zatem do tego doszło. Chcieliśmy wstawić do hashmapy klucz k, lecz w wiaderku $\mathcal{H}(k)$ napotkaliśmy element l. Rozpatrzmy dwa sposoby rozwiązania tego kryzysu.

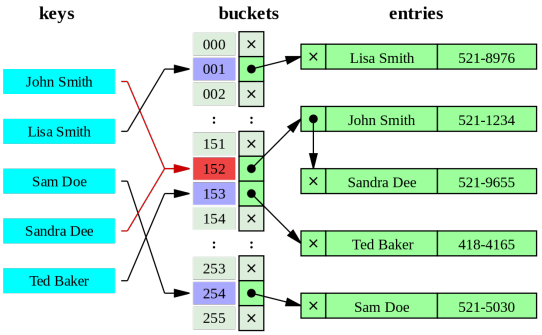
1.2.1 Adresowanie otwarte – Open addressing

W tym sposobie udajemy, że w sumie to nic złego się nie stało, i możemy po prostu tymczasowo lekko zmodyfikować naszą funkcję hashującą. Na przykład możemy spojrzeć, czy przypadkiem następne wiaderko (o indeksie $\mathcal{H}(k)+1$) nie jest puste, i tam wstawić element. Korzystając z tego sposobu implementacja może być bardziej skomplikowana, ale zwykle osiągniemy lepszą wydajność.

[1] Obrazek pochodzi z Wikimedia Commons: `https://commons.wikimedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg`

1.2.2 Łańcuchy – Collision chaining

W tym sposobie w każdym wiaderku skonstruujemy listę, do której będziemy dodawać kolejne elementy, które trafiły do tego wiaderka[2]. Aby zaoszczędzić na pamięci, może być to najprostsza lista kierunkowa. Tak więc w tym wypadku w $\mathcal{H}(k)$ znajdziemy listę $l \to \emptyset$, do której dodamy k, otrzymując $l \to k \to \emptyset$.



1.3 Podsumowanie

Jako funkcji hashujących często używa się takich wyrażeń jak wielomiany niskiego stopnia modulo liczba pierwsza n, która przy okazji jest liczbą wiaderek. Wykorzystuje się także liczne chaotyczne operacje bitowe.

Problemem w implementacji jest skuteczna adaptacja liczby wiaderek do obecnego stanu struktury. Podobnie jak w dynamicznych tablicach, w pewnym momencie należy dokonać realokacji (rehashowania), w którym przepisujemy wszystkie elementy do większej liczby wiaderek. W tym celu wyznacza się ułamek będący maksymalnym stopniem zapełnienia wiaderek. W praktyce zapisuje się listę liczb pierwszych, które są możliwymi liczbami wiaderek.

Często spotykanym problemem jest też to, że klucze będące liczbami całkowitymi są przed hashowaniem niemądrze modulowane przez 2^{32}. Tak też w standardowej implementacji hashmap w STLu z `libstdc++` można stworzyć przypadek pesymistyczny $O(n)$ na każdą operację wrzucając do struktury wielokrotności 2^{32} – bo wtedy wszystkie liczby trafią do tego samego wiaderka.[3]

Podsumowując, hashmapy są przydatną strukturą, bo napisane w odpowiedni sposób są o wiele prostsze od BST (które zwykle służą do implementowania zbiorów i słowników) i bywają szybsze. Najczęściej hashmapa będzie już zaimplementowana w języku, z którego korzystamy – warto jednak kojarzyć, jak działają. O tworzeniu hashy można przeczytać tutaj: `http://ticki.github.io/blog/designing-a-good-non-cryptographic-hash-function/`

[2] Obrazek poniżej również pochodzi z Wikimedia Commons: `https://commons.wikimedia.org/wiki/File:Hash_table_5_0_1_1_1_1_LL.svg`

[3] Dotyczy to 64-bitowych liczb na 32-bitowej architekturze. Bądźcie łaskawi wobec osób, które zhakujecie w ten sposób.

2 Kopiec randomizowany

Kopce to grupa struktur drzewiastych przechowujących zbiór wartości, które spełniają własność kopca: dla każdego wierzchołka p i jego dziecka c, zachodzi $v_p \geqslant v_c$ (gdzie v to wartość przypisana do wierzchołka). Najczęściej implementuje się kopce binarne (w których przechowywane jest drzewo binarne), które są najprostsze i dzięki temu często najszybsze. Jednak istnieje wiele innych kopców, różniących się złożonością i możliwymi operacjami.

My będziemy rozważać kopiec randomizowany[4], który swoją budową przypomina kopiec binarny, lecz zasada jego modyfikacji opiera się na losowości. Nie jest on wiele wolniejszy od zwykłego kopca, a pozwala na łączenie dwóch kopców w $O(\log n)$, co może okazać się przydatną operacją, która nie jest możliwa z kopcami binarnymi.

2.1 Łącz i zwyciężaj

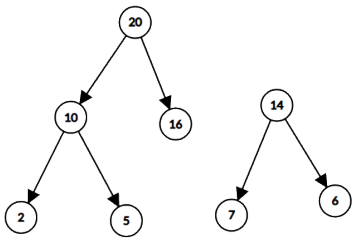
Kopiec randomizowany inspirowany jest kopcem binarnym – wierzchołki będą układać się w drzewo binarne. Każdy wierzchołek ma lewe oraz prawe dziecko. Będziemy też chcieli wykonywać te same główne operacje, co kopiec binarny:

- `top()` – zwróć największy element (znajdujący się w korzeniu).
- `pop()` – usuń największy element (wierzchołek będący korzeniem).
- `push(x)` – dodaj x do kopca.

Operacje `top()` to kwestia sprawdzenia wartości w korzeniu, więc nie musimy jej dodatkowo rozważać. Skupmy się na `pop` i `push`. Zauważmy, że znając sposób łączenia kopców `merge(A, B)` moglibyśmy wykonywać operacje w następujący sposób:

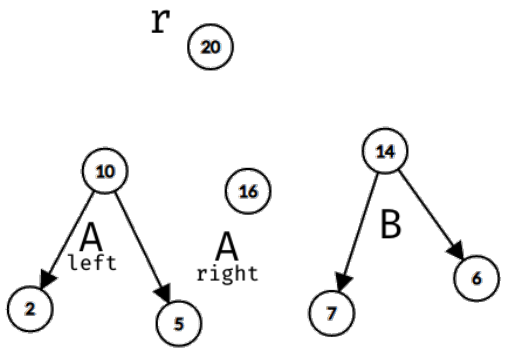
- `pop()`: `merge(root->left, root->right)`
- `push(x)`: `merge(root, {x})` (`{x}` to wierzchołek o wartości x).

Wystarczy się teraz zastanowić, jak efektywnie łączyć dwa kopce binarne. Chcemy przy tym, by powstały kopiec był jak najbardziej zbalansowany, czyli żeby jego wysokość była jak najmniejsza.



[4]Na podstawie: `https://cp-algorithms.com/data_structures/randomized_heap.html`

Łączymy dwa kopce A i B chcąc stworzyć C. Nietrudno zauważyć, że korzeniem C musi być korzeń A lub B – ten o większej wartości. Odłączmy zatem bardziej wartościowy z korzeni (bez straty ogólności załóżmy, że był to korzeń A) i oznaczmy go r.



W ten sposób powstały nam trzy drzewa: A_{left}, A_{right} oraz B. Załóżmy, że dotychczas kopiec był rozsądnie zbalansowany i wysokość poddrzew z A jest podobna (w optymalnym kopcu wysokość obu z nich jest logarytmiczna od ich wielkości), ale niekoniecznie muszą one być podobne do wysokości B.

Musimy jakoś złączyć jedną parę tych drzew w jedno, a następnie przypisać pozostałe dwa jako dzieci r. Tak też moglibyśmy złączyć A_{left} i A_{right}, ale wtedy B może mieć kompletnie inną wysokość psując zbalansowanie drzewa. Połączmy zatem B z A_{left} *lub* A_{right}. Ponieważ decyzja, z którym drzewem połączyć B, jest bardzo trudna, wylosujmy to poddrzewo. Bez straty ogólności załóżmy, że wylosowaliśmy A_{left}. Pozostaje teraz rekurencyjnie złączyć A_{left} i B i przypisać je jako dziecko r. Drugim dzieckiem r jest A_{right}.

W praktyce wystarczy napisać funkcję losującą bit (liczbę 0 lub 1), a implementacja będzie bardzo prosta:

```
node* merge(node* first, node* second)
{
    if(not first or not second)
        return first ? first : second;
    // `first` to nowy korzeń
    if(first->value < second->value)
        swap(first, second);
    if(next_bit())
        swap(first->left, first->right);
    // `right` zostawiamy, `left` łączymy
    first->left = merge(first->left, second);
    return first;
}
```

2.2 Praktyka

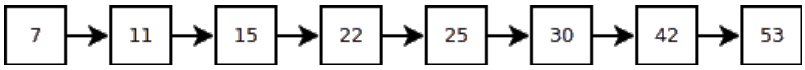
Choć pomysł kopca randomizowanego jest bardzo ciekawy, trudno znaleźć zadanie, w którym jest on wymagany. Możemy jednak rekreacyjnie poszukać jego dodatkowych możliwości: zauważmy na przykład, że możemy łatwo dodać prostą opóźnioną propagację pozwalającą na dodawanie pewnej wartości Δ do wszystkich wartości w danym kopcu, co będzie respektowane przy `merge`.

3 Skiplista

Skiplista to struktura bazującą na listach kierunkowych. Przechowuje uporządkowany ciąg elementów. Tak na prawdę jest to po prostu lista ze sprytnie dobranymi skokami (*skip*), przypominającymi jump pointery, które pozwalają na przeskoczenie wielu następnych elementów.

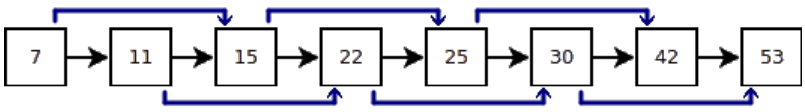
3.1 Skoki

Zacznijmy od próby rozszerzenia zwykłych list kierunkowych do praktycznej struktury pozwalających na szybkie dodawanie, usuwanie oraz znajdowanie elementu w strukturze (podobne operacje, co hashmapa). Chcemy, by lista utrzymywała elementy w kolejności posortowanej.[5]



Wykonywanie operacji zależy od złożoności operacji znalezienia ostatniego elementu mniejszego od danego x. (bo potrzebna jest nam by znaleźć miejsce na dodawany element lub pozycję, gdzie znajduje się ten usuwany). W zwykłej uporządkowanej liście kierunkowej, z którą teraz mamy do czynienia, ta operacja ma złożoność $O(n)$ (gdzie n to liczba elementów).

Pojawia się prosty pomysł inspirowany jump pointerami: dla każdego wierzchołka przechowujmy dodatkowy wskaźnik pozwalający na przeskoczenie k elementów.



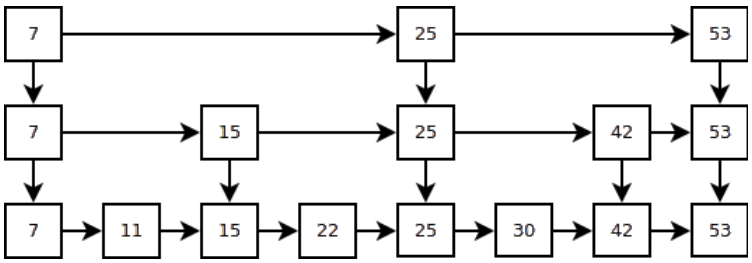
Chcąc teraz wstawić lub usunąć element, musimy zaktualizować $O(k)$ skoków, które go pomijają. Chcąc znaleźć ostatni mniejszy element, mniejszy co

[5] Większość obrazków (niektóre są zmodyfikowane) pochodzi z artykułu z `http://ticki.github.io`. Link jest na końcu sekcji.

najwyżej $O(k)$ razy skoczymy o jeden, a skoki wykorzystamy $O(\frac{n}{k})$ razy. Inspirując się pierwiastkami można zauważyć, że optymalnie jest dobrać $k = \sqrt{n}$, co da nam taką samą złożoność na operacje. Pozostaje kwestia tego, że musimy te wskaźniki odpowiednio odbudowywać, gdy optymalne k zmienia się, co utrudnia sprawę[6]. Budowanie klasycznych jump pointerów kompletnie odpada, bo trzeba wtedy aktualizować zbyt dużo wartości. Należy z tego wyciągnąć taki wniosek, że utrzymywanie skoków konkretnej wielkości jest nieopłacalne, i musimy pozwolić na pewne fluktuacje.

3.2 Wyższy poziom

Można wykorzystać inną ideę, niż zwykłe wskaźniki o konkretną liczbę elementów. W tym celu należy pomyśleć dwuwymiarowo. Nadajmy każdemu wierzchołkowi w liście poziom (*level*), domyślnie równy zero, i powiedzmy, że każdy wierzchołek ma wskaźnik do poziomu niżej oraz do następnego wierzchołka o tym samym (lub większym) poziomie – w tym układzie wierzchołki poziomu zero to zwykła jednowymiarowa lista kierunkowa nieświadoma istnienia wyższych poziomów. Warto zauważyć, że w tym układzie jeden element może być w wielu wierzchołkach.



Moglibyśmy teraz zrobić coś sprytnego, myśląc o takiej poziomowanej liście jako statycznej, tworząc perfekcyjną strukturę, w której do każdego elementu można się dostać w $O(\log n)$ kroków. Jednak wymyślenie takiego układu jest trudne, i do tego nieopłacalne (bo robimy dynamiczną strukturę – moglibyśmy przypadkiem popsuć układ usuwając kluczowe elementy o wysokich poziomach).

Zamiast tego przydzielajmy poziomy *losowo*. Gdy dodajemy element, jednocześnie przydzielamy mu pewien poziom l. Zastanówmy się, jaka powinno być szansa na otrzymanie każdego poziomu. Chcielibyśmy, by elementy na najwyższym poziomie pozwalały nam skakać mniej więcej o połowę ciągu. Na drugim: o ćwierć, na trzecim: o jedną ósmą, et cetera. Dzięki temu, łatwo możemy doskoczyć do poszukiwanego elementu w $O(\log n)$. Dochodzimy do wniosku, że chcielibyśmy, by na poziomie $l + 1$ było o połowę mniej wierzchołków niż na l. Stąd pomysł, by prawdopodobieństwem $p(l)$ na przydzielenie poziomu l było:

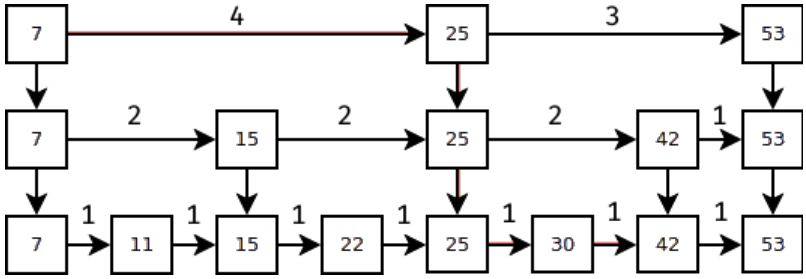
$$p(l) = \left(\frac{1}{2}\right)^{l+1}$$

[6]Można to jednak zrobić tak, by osiągnąć amortyzowane $O(\sqrt{n})$ na wszystkie operacje. Czy umiesz to zrobić, czytelniku?

Dzięki temu, że układ poziomów jest teraz losowy, właściwie niemożliwe jest jego przypadkowe popsucie. Warto zauważyć, że ten rozkład można symulować przez zliczanie rzutów monetą dopóki nie wyrzucimy pierwszego orła.

3.3 Podsumowanie

Taką strukturę możemy w prosty sposób rozszerzyć, by implementowała operacje *order statistics* oraz pozwalała na liczenie prostych operacji na prefiksie. Na przykładzie order statistics: chcemy umieć znajdować k-ty co do wielkości element oraz liczyć, który z kolei jest dany element w strukturze (ile jest elementów mniejszych). W tym celu dla każdego skipa będziemy przechowywać, ile elementów przeskakujemy. Przy dodawaniu i usuwaniu elementów zmieniają się tylko sąsiednie skipy, więc modyfikacja metadanych przebiega w czasie stałym. Samo szukanie k-tego elementu to kwestia sprytnego wyszukiwania binarnego, a liczenie elementów mniejszych od danego jest analogiczne do operacji znalezienia konkretnego elementu, przy czym musimy zliczać, ile elementów skipowaliśmy. Na rysunku poniżej przedstawione jest takie rozszerzenie i ścieżka poszukiwań 6-tego (od 0) elementu.



Więcej o implementacji, zastosowaniach, plusach i minusach struktury można przeczytać w artykule oraz na Wikipedii. Zwykle skiplisty porównywane są do BST, bo implementują podobne operacje. Niestety, skiplisty są trochę mniej elastyczne. Jednak ich implementacja jest o wiele prostsza od zaawansowanych BST.

Wikipedia:

`https://en.wikipedia.org/wiki/Skip_list`

Artykuł o implementacji:

`http://ticki.github.io/blog/skip-lists-done-right/`

Praca o dodatkowych operacjach:

`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.524&rep=rep1&type=pdf`

4 Moc i sekretarka

Warto jeszcze krótko nadmienić o problemach, których rozwiązanie jest bardzo proste, a ich główną częścią jest pewna matematyczna własność.

Zaokrąglanie wielkości zbioru W pierwszym problemie mamy dany zbiór o nieznanej wielkości n. Znamy sposób wybrania losowego z niego elementu. Naszym celem jest podanie zaokrąglenia na jego wielkość wykonując możliwie małą liczbę sprawdzeń.

Wystarczy sobie przypomnieć paradoks dnia urodzin. Pierwszy raz wylosujemy ten sam element drugi raz po około

$$k = \sqrt{\frac{\pi n}{2}}$$

(Jest to dokładniejsze zaokrąglenie, niż to pokazane na pierwszej kartce). Zatem otrzymujemy prosty algorytm: losujemy elementy, zapisując je w jakimś zbiorze, i gdy napotkamy wcześniej znaleziony element zwracamy wynik:

$$n \approx \frac{2k^2}{\pi}$$

Jeżeli chcemy otrzymać dokładniejszą aproksymację, możemy powtórzyć algorytm wielokrotnie.

Problem sekretarki Mamy n kandydatów (i kandydatek) na posadę sekretarza (lub sekretarki). Każdy z nich przychodzi kolejno na rozmowę kwalifikacyjną i ma kwalifikacje, które możemy wyrazić pewną liczbą całkowitą (wszystkie te liczby są parami różne). Nie znamy kwalifikacji kandydata, zanim nie przejdzie on rozmowy kwalifikacyjnej. Chcemy wybrać **najlepszego** z kandydatów pod względem kwalifikacji, lecz kandydata możemy wybrać tylko przed następną rozmową i nigdy więcej. Sukcesem jest sytuacja w której wybraliśmy najlepszego kandydata (i żadna inna). Podaj algorytm maksymalizujący szansę powodzenia.

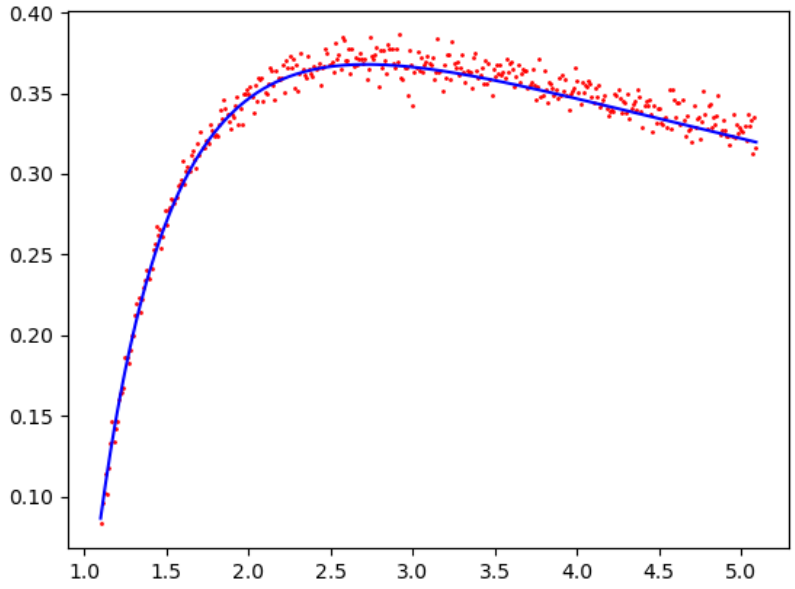
Spróbujmy wykorzystać prosty pomysł: weźmy pierwsze $\frac{n}{f}$ kandydatów, wyznaczmy wśród nich maksymalne kwalifikacje (punkt odniesienia), a następnie spośród pozostałych wybierzmy pierwszego wykwalifikowanego bardziej niż dotychczasowe maksimum. Pozostaje wyznaczyć współczynnik f i znaleźć taki, dla którego szanse powodzenia są największe.

Spróbujmy znaleźć go metodą prób i błędów dla $n = 100$.

f	Szansa powodzenia
2	34.87%
1.5	27.05%
3	36.89%
2.5	36.99%
2.75	37.12%
2.71	37.13%

Można się domyśleć, że najlepszą stałą jest $f = e$. Staje się to jasne, gdy spojrzy się na wykresik. Można się wtedy łatwo przekonać, że sama funkcja to

$$p(f) = \frac{\log f}{f}$$



Rysunek 1: Na czerwono: prawdopodobieństwo powodzenia względem f, $n = 100$ po 5000 prób. Na niebiesko: funkcja $p(f) = \frac{\log f}{f}$.

5 Algorytm Freivaldsa

$$\boldsymbol{AB} \stackrel{?}{=} \boldsymbol{C} \to O(n^3)$$

$$\boldsymbol{AB} = \boldsymbol{C} \implies \boldsymbol{AB}\vec{w} = \boldsymbol{C}\vec{w} \quad (w_i \in \{0,1\})$$

$$\boldsymbol{AB}\vec{w} = \boldsymbol{A}(\boldsymbol{B}\vec{w}) \stackrel{?}{=} \boldsymbol{C}\vec{w} \to O(n^2)$$

$$p = \frac{1}{2}$$

Algorytm Freivaldsa to algorytm weryfikacji poprawności mnożenia macierzy. Losuje on wektor $\vec{w}$ składający się z 0 i 1 i wykorzystuje to, że mnożenie macierzy jest łączne, redukując złożoność z $O(n^3)$ do $O(n^2)$. Równość iloczynów z $\vec{w}$ niekoniecznie implikuje, że iloczyn macierzy też jest równy (implikacja w jedną stronę). Jeżeli otrzymamy odpowiedź, że $\boldsymbol{AB}\vec{w} \neq \boldsymbol{C}\vec{w}$, to na pewno wiemy, że tak jest. Jednak jeżeli dostaniemy odpowiedź twierdzącą, to jest ona poprawna z prawdopodobieństwem $\frac{1}{2}$. Możemy wykonać algorytm wielokrotnie, aby zwiększyć to prawdopodobieństwo.