Dokumentacja AAL

„Sortowanie kulek na pochylni za pomocą dźwigu przenoszącego trzy na raz”

Michał Sobieraj 300269 **Michał Łątkowski 300208**

Treść problemu:

Na pochylni są ułożone w przypadkowej kolejności kulki w trzech kolorach: czerwone, zielone i niebieskie. Należy je uporządkować za pomocą dźwigu w taki sposób, aby na początku (najniżej) były czerwone, następnie zielone, a na końcu (najwyżej) niebieskie. Dźwig w jednym ruchu może podnieść do góry jednocześnie trzy sąsiednie kulki z dowolnego miejsca na pochylni i przenieść je na koniec, za kulki leżące powyżej, które stoczą się pod własnym ciężarem w dół, wypełniając lukę. Ułożyć plan pracy dźwigu, dyktujący w kolejnych ruchach, którą trójkę kulek przenieść na koniec. Wykonać jak najmniejszą liczbę ruchów.

Założenia:

Założyć, że na pochylni są przynajmniej 3 kulki niebieskie. Zakładamy też, że liczba kulek w poszczególnych kolorach będzie znana lub robot będzie mógł ją wyliczyć

Narzędzia:

Kod w języku C++ używając kompilacji z użyciem Makefile by zapewnić działanie na obu wymaganych systemach operacyjnych (Windows i Linux)

Działanie programu:

Program działa w 3 trybach opisanych w poleceniu, umożliwiając zarówno wczytanie danych z pliku za pomocą przekierowania wejścia w shellu, jak i generację parametryczną problemu czy też przeprowadzenie pełnego cyklu testowego o zadanych parametrach.

Generacja pseudolosowych wartości opiera się na bibliotece <random> i wbudowanym weń algorytmie Mersenne Twister.

Pomiar czasu opiera się na bibliotece <chrono>.

Program nie posiada interfejsu graficznego i jest wywoływany za pomocą poleceń konsolowych.

Algorytmy:

Algorytm 1:

Algorytm po kolei wyszukujący kulki obecnie porządkowanego koloru i przesuwający je na umowny początek obszaru nieposortowanego, który z początku jest równy najniższemu miejscu na pochylni a w miarę sortowania rośnie.

Jeśli do posortowania zostało **5** lub **4** kulek, mamy skończoną liczbę scenariuszy (odpowiednio **5** i **4**) na ustawienie jej na umowny początek, w zależności od jej pozycji.

W przeciwnym wypadku, jeśli nie jest już ustawiona, ustawiamy kulkę tak by była na miejscu **n** od umownego początku, takim że **n** mod **3** = **1** i wybieramy kulki od umownego początku dopóki nie znajdzie się ona na właściwym miejscu. To ustawienie zależy od podzielności przez **3** liczby kulek nieposortowanych. Po posortowaniu wszystkich kulek koloru czerwonego, algorytm wywołujemy jeszcze raz dla koloru zielonego.

Algorytm 2:

Algorytm 2 zamiast pojedynczo przestawiać kulki na początek opiera się na grupowaniu kulek trójkami by zmniejszyć ilość przeniesień. Wyróżniamy dwa typy kulek: kulki koloru, który chcemy sortować oraz kulki innych kolorów.

W początkowej fazie algorytmu na początek przekładamy **0**, **1** lub **2** kulki szukanego koloru oraz **1** lub **2** kulki innych kolorów, w zależności od reszty z dzielenia ilości kulek innych kolorów przez **3**, po czym układamy **2** kulki szukanego koloru (potrzebne do ostatniego kroku). Układamy je tak jak to robiliśmy w algorytmie 1.

Następnie za pomocą zmiennych pomocniczych head oraz tail przeszukujemy od końca pochylnię rozpatrując trójki kulek. Zmniejszamy head o **3** (idąc od końca w kierunku początku). Jeśli wśród kulek jest choć jedna koloru który chcemy sortować, przenosimy wszystkie na koniec i zmniejszamy tail o **3**, zliczając przy okazji ilość kulek innych kolorów. Gdy ilość ta przekroczy **3**, kulki będące za miejscem tail są sortowane algorytmem 1 układającym na początku kulki innego koloru, a tail zwiększamy o **3**.

Powtarzając te kroki otrzymamy układ kulek:
|specjalnie ułożony początek|---|Trójki kulek innego koloru|--|kulki koloru sortowanego|

W tym momencie jeśli liczba kulek innych kolorów była niepodzielna przez **3** przesuwamy **3** pierwsze kulki na koniec, po czym przesuwamy kulki od miejsca **3** (pierwsze dwa są już ułożone), dopóki kulki koloru sortowanego nie zsuną się na początek.

Algorytm potem powtarzamy dla drugiego koloru.

Algorytm 3:

Algorytm 3 polega na tworzeniu drzewa rozwiązań, dodając do wektora pomocniczego wszystkie możliwe ruchy w odniesieniu do aktualnego stanu, w ten sposób przeszukujemy "poziomo" drzewo rozwiązań.

Algorytm 4:

Algorytm działa podobnie do algorytmu 1, ale priorytetyzuje przeniesienie na początek najdłuższych ciągów zamiast przenosić kulki po kolei.

Tryby działania programu:

1) crane -m1 <in.txt >out.txt
W tym trybie in.txt i out.txt to odpowiednio plik, z którego pobieramy dane dot. kulek (ciąg literek R, G i B) oraz plik, do którego zapisujemy ilość ruchów wykonanych przez nasz algorytm. in.txt musi być podane, ale out.txt może nie być, wtedy program wypisze dane (czas, jaki zajęło wykonanie algorytmu i ilość ruchów) na konsolę.

2) crane -m2 -n X -d B G R >out.txt
W tym trybie X to ilość kulek a B, G, R to ilości kulek odpowiednich kolorów (program generuje ciąg kulek proporcjonalnie i go miesza). B+G+R musi równać się X. W przypadku niepodania B, G i R (czyli polecenia w formie crane -m2 -n X >out.txt) algorytm wygeneruje losowy ciąg kulek o długości X. Do out.txt, jeśli zostanie podane, algorytm wypisuje czas jaki zajęło wykonanie algorytmu i ilość ruchów

3) crane -m3 -n A -k B -step C -r D
Ten tryb pozwala na generację pełnego ciągu testowego. A to początkowa ilość kulek, B to ilość problemów, C to krok, D to ilość instancji problemu (np. dla A=500, B=3, C=500, D=10 program analizuje problemy o rozmiarze 500, 1000 i 1500, dla każdego rozmiaru problemu generując 10 instancji problemu). Program zapisuje wyniki testów w pliku output_table.txt

4) crane -m4 -n A -p B >out.txt
Ten tryb pozwala na generację takiego ciągu kulek o rozmiarze A, że istnieje prawdopodobieństwo B (dopuszczalne wartości od 0 do 1), że każda (prócz oczywiście pierwszej) kulka będzie taka sama, jak kulka poprzednia. W przeciwnym wypadku kulka będzie wygenerowana z równą szansą na każdy z trzech kolorów. Do out.txt, jeśli zostanie podane, algorytm wypisuje czas jaki zajęło wykonanie algorytmu i ilość ruchów

Dostępne algorytmy:

1) Najlepszy algorytm, który dzięki opisanym wyżej w sekcji "Algorytmy" (algorytm 2) modyfikacjom pozwala na rozwiązanie problemu w czasie liniowym

2) Gorszy algorytm (algorytm 1), którego złożoność jest w teorii kwadratowa.

3) Rozwiązanie brutalne, przeszukujące drzewo rozwiązań

4) Zmodyfikowana wersja problemu, gdzie dźwig priorytetowo traktuje długie ciągi monokolorowe

Struktura programu:

Program składa się z plików: Clock.cpp, Generator.cpp, InterfaceHandler.cpp, AlgorithmRunner.cpp i main.cpp (wszystkie prócz main mają swoje pliki nagłówkowe .hpp).

-Clock implementuje zegar, udostępnia możliwość rozpoczęcia mierzenia czasu, zakończenia mierzenia czasu i wyświetlenia czasu, który upłynął pomiędzy wywołaniem metody rozpoczęcia i zakończenia pomiaru

-Generator pozwala na cztery typy generacji: całkowicie losową, sparametryzowaną (tzn. użytkownik podaje ile ma być kulek o danym kolorze), probabilistyczną (tzn. użytkownik podaje z jakim prawdopodobieństwem każda kulka prócz pierwszej ma być tego samego koloru co poprzednia) i z pliku (używane w przypadku wybrania trybu 1).

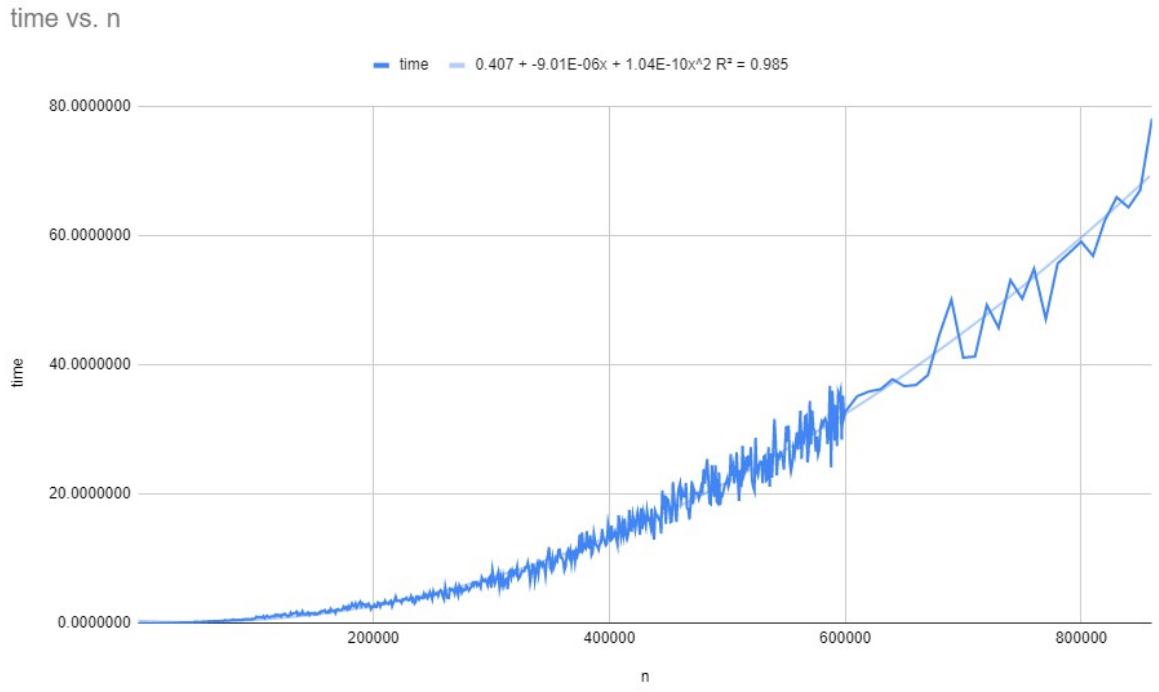
-InterfaceHandler zajmuje się wczytaniem argumentów podanych przed użyciem, sprawdzeniem jej poprawności, pobraniem od użytkownika informacji o wybranym algorytmie i uruchomienie odpowiedniego.

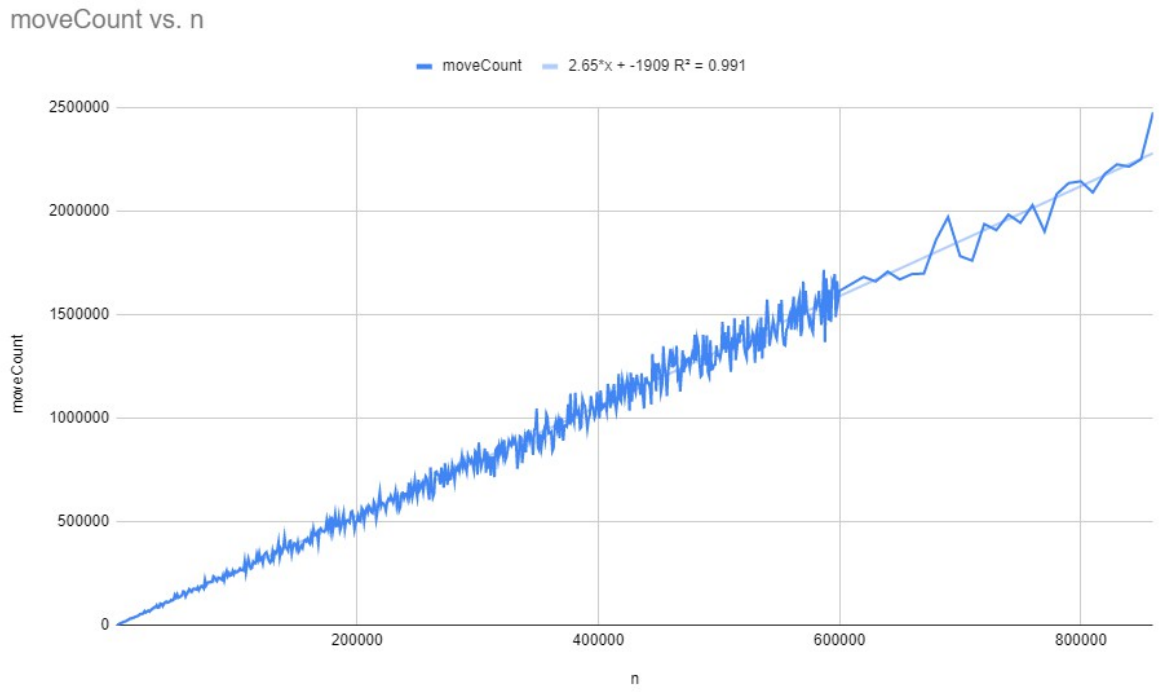
-AlgorithmRunner to klasa przechowująca wszystkie metody odpowiadające za sortowanie ciągu kulek i informowanie użytkownika o wynikach sortowania

Wyniki i wnioski:

(link do formularza z danymi https://docs.google.com/spreadsheets/d/1aN1mqvnsGZE7zs5Pz-nw_-8pTO7-N5u_iicfWigBX8U/edit?usp=sharing)

ALGORYTM 1



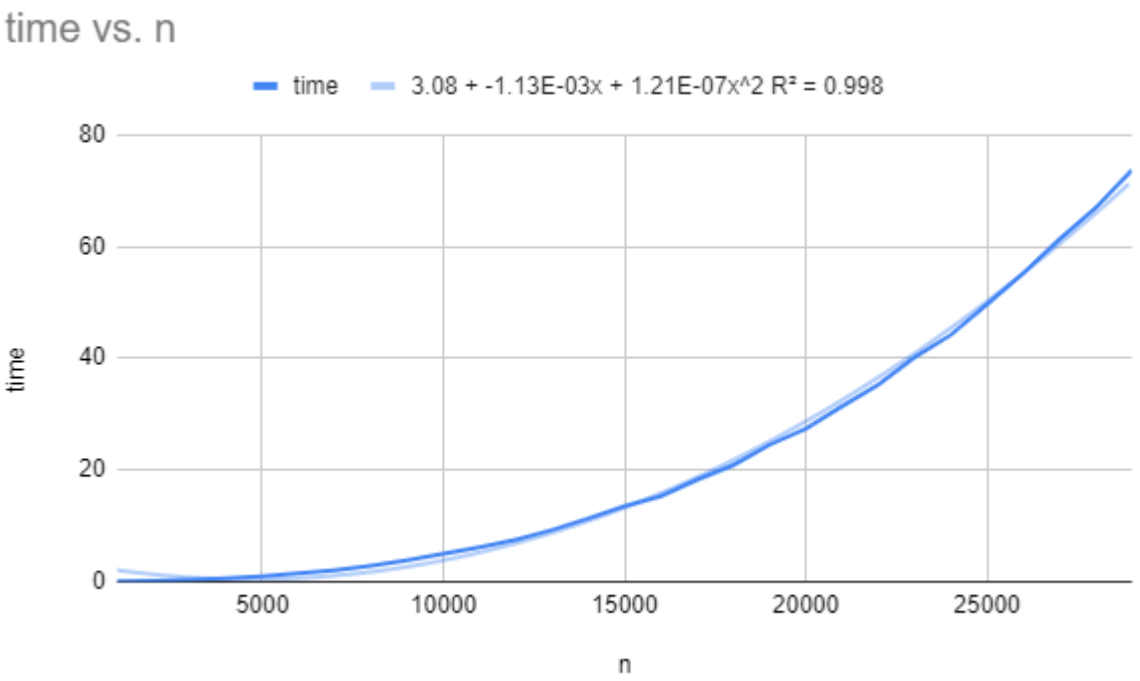


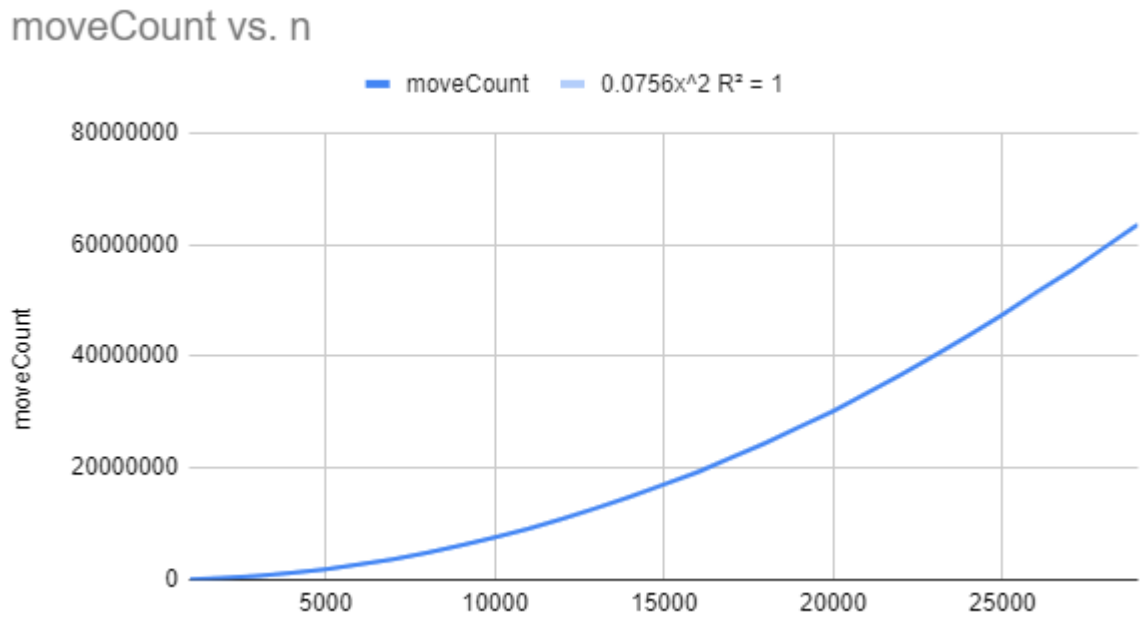
Widać, że liczba ruchów jest liniowa, co potwierdza tezę o potencjalnej liniowości samego algorytmu. Sam czas jednak liniowy nie jest i najlepiej jest aproksymowany opisaną na wykresie funkcją wielomianową stopnia 2 (wolne "wzrastanie" tej funkcji wynika z niezwykle małych współczynników przy x^2 i x). Niemniej jednak uważamy, że wynika to ze szczegółów implementacyjnych, mianowicie użycia wektora. Funkcja erase() dostępna w STL-owym wektorze wprowadza złożoność O(n).

Teoretycznie opracowanie całego rozwiązania na listach, a nie na wektorze, w idealnym przypadku dałoby złożoność liniową. Należy jednak dodać, że nie jesteśmy pewni, czy dodatkowe operacje na iteratorach, które wówczas byłyby potrzebne, nie sprawiłyby, że ta złożoność i tak nie będzie w praktyce liniowa.

Przeprowadzaliśmy testy do momentu, gdy jedno wykonanie przekraczało 60 sekund (przy 10 próbkach jeden obieg dla danej wartości N trwał 6 minut). Warto zwrócić uwagę jak małe współczynniki przy x i x^2 wpłynęły na kształt wykresu-gdyby nie przeprowadzać testów aż do wartości typu 800000, tylko np. do 100000, można by odnieść mylne wrażenie, że algorytm ma złożoność liniową.

ALGORYTM 2

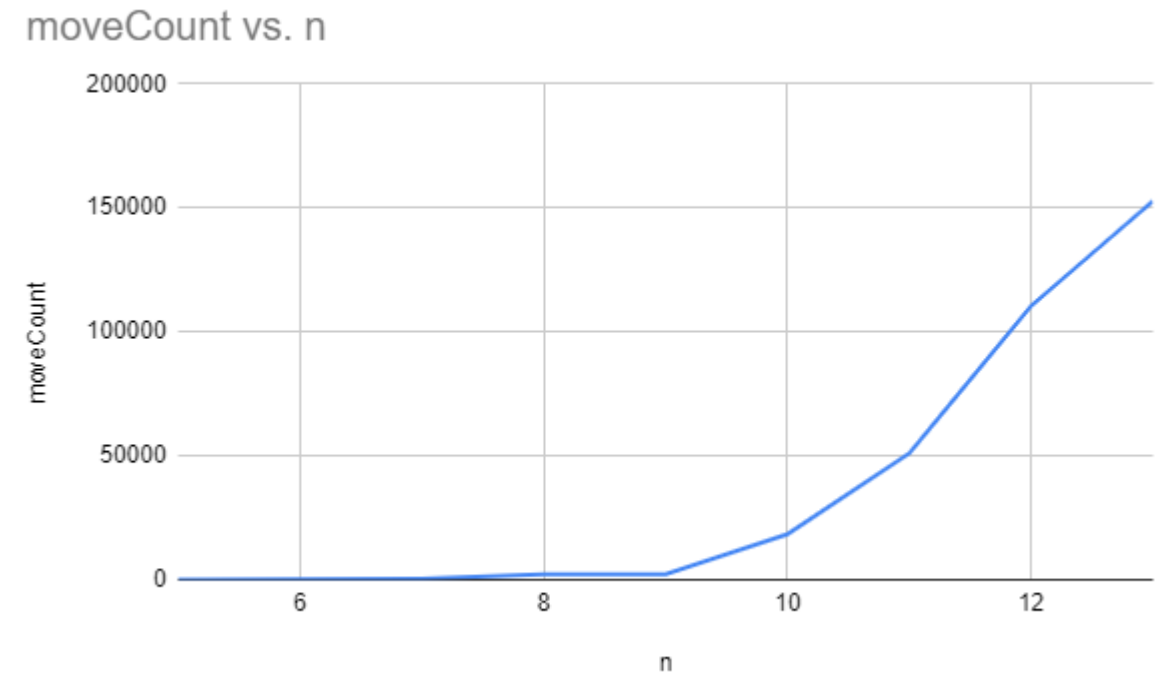


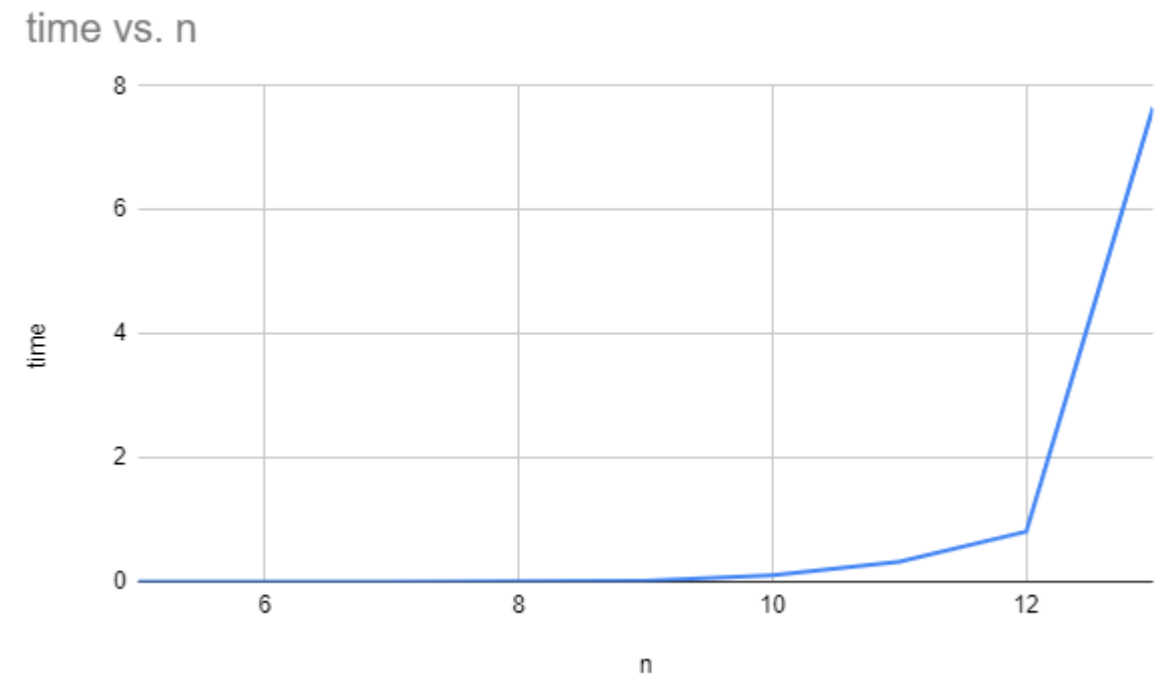


Tutaj kwadratowa złożoność rozwiązania jest już bardziej widoczna, uwydatnia się na dużo mniejszych ilościach kulek (dla 25000 kulek sortowanie trwa około 50 sekund, dla porównania algorytm 1 poradziłby sobie z tym rozmiarem problemu w 0.05 sekundy). Widoczna jest ona zarówno w liczbie ruchów, jak i w czasie, co wyraźnie wskazuje na zgodność przewidywań teoretycznych z praktyką.

Nie ma więc najmniejszych wątpliwości, że dla każdych zastosowań algorytm 1 ma wyraźną przewagę-jednakże porównanie ich obu na jednym wykresie jest awykonalne poprzez zbyt duże dysproporcje w czasach wykonania sortowania.

ALGORYTM 3



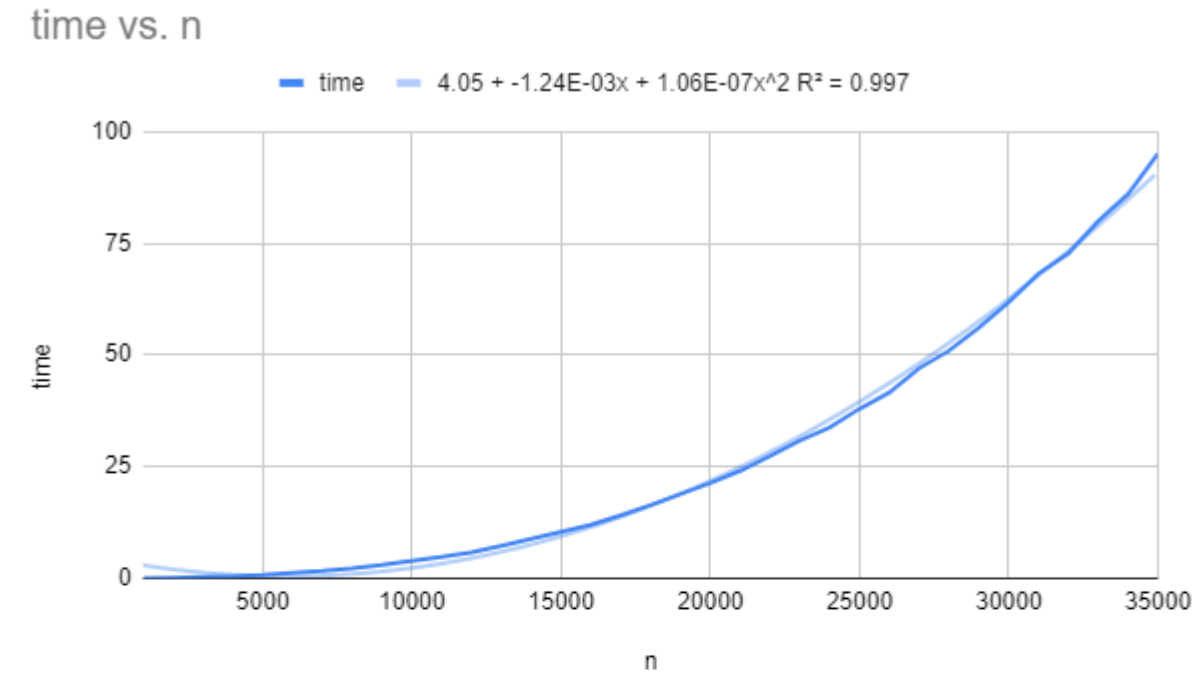


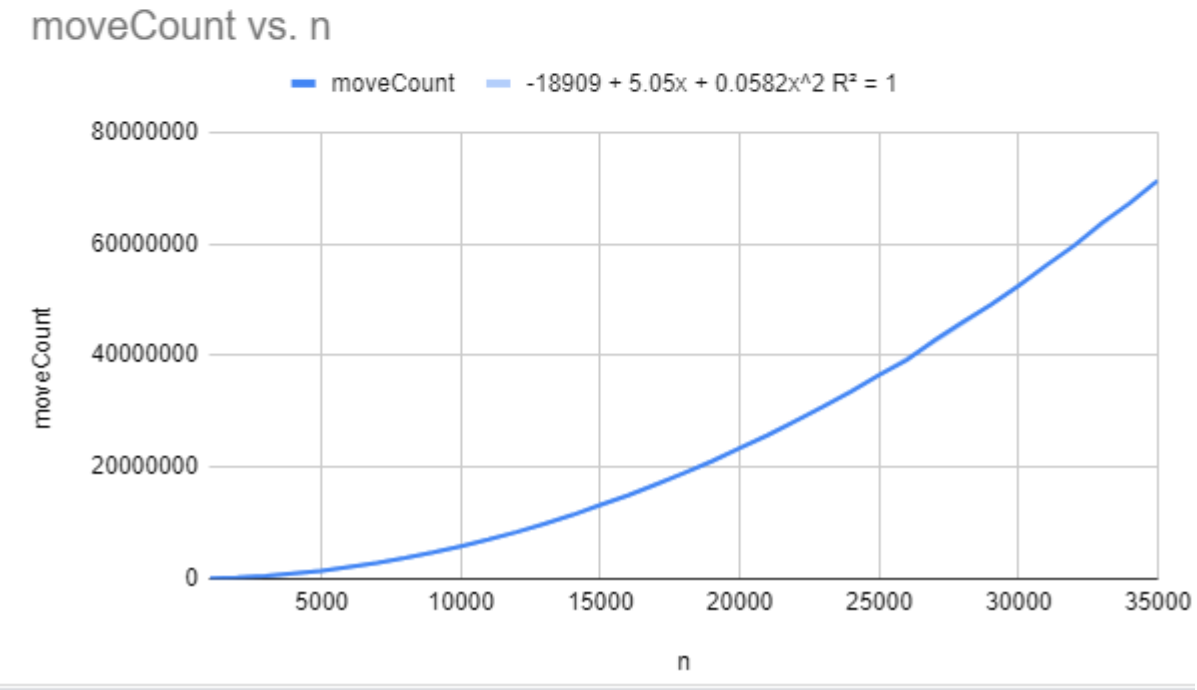
ALGORYTM 3

Algorytm 3, czyli brutalne przeszukiwanie drzewa rozwiązań, okazuje się dla naszego problemu algorytmem nie tylko skrajnie niewydajnym, ale również praktycznie niedziałającym. Dla większych rozmiarów problemu drzewo rozrasta się do takich rozmiarów, że występują problemy z alokacją pamięci w strukturach STL. Dla rozmiarów około 14 kulek, program dla 10 próbek nie znajduje ani razu rozwiązania przed wyświetleniem wyjątku *bad_alloc* (który przechwytywaliśmy). Zachowanie

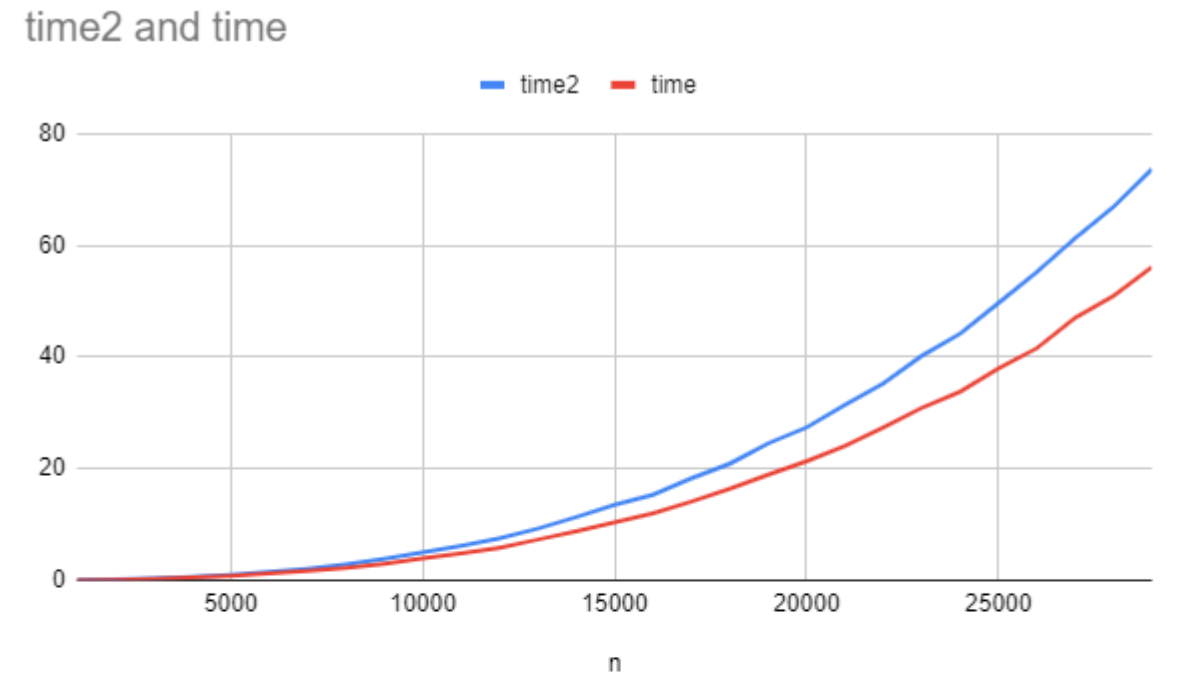
programu dla większych rzędów ilości kulek (np. 100000) można jedynie zatem ekstrapolować z faktu, że już dla 13 kulek potrzebował ponad 7 sekund.

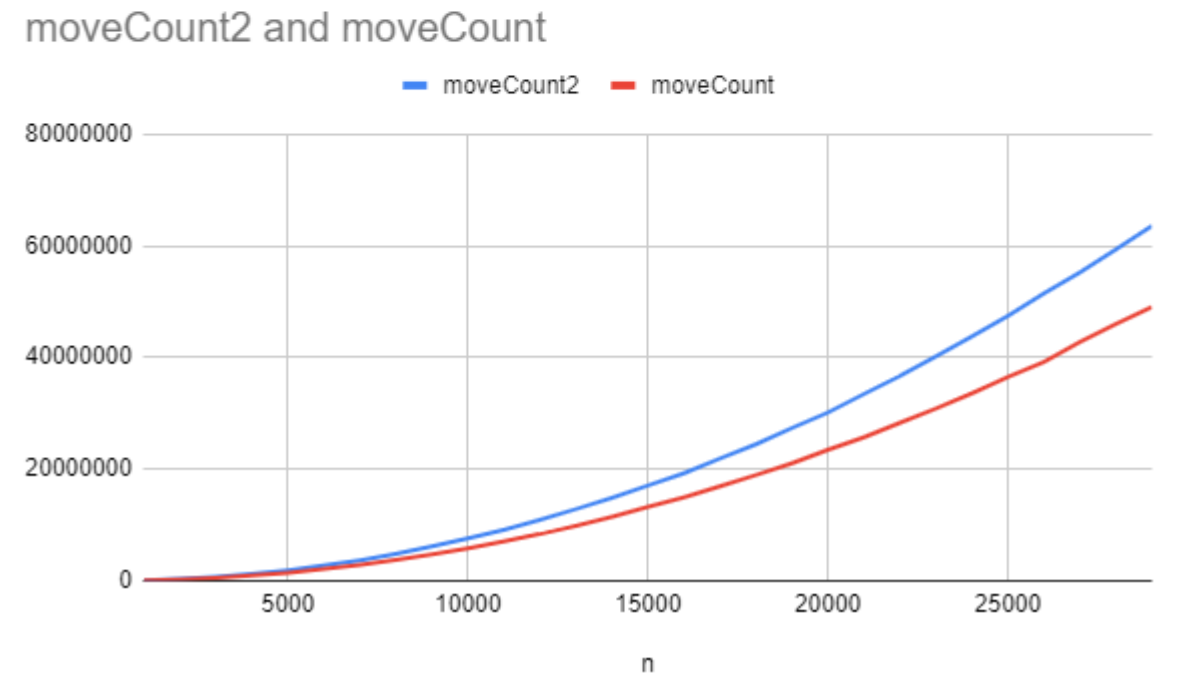
ALGORYTM 4





Wprowadziliśmy też zgodnie z zaleceniami algorytm 4, w którym dźwig największy priorytet kładzie na długich ciągach monokolorowych kulek. Zgodnie z przewidywaniami, jego wyniki są bliższe algorytmowi 2 niż znacznie bardziej wydajnemu algorytmowi 1.





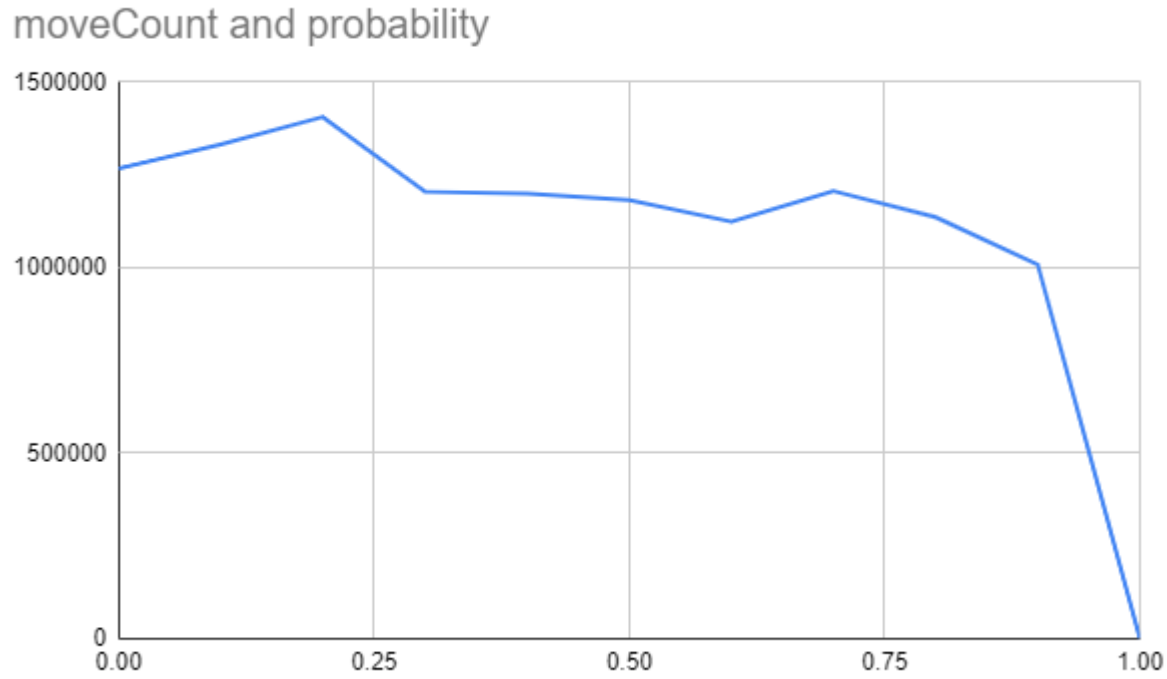
Jak widać wyżej, algorytm 4 (czerwony) można porównać z algorytmem 2 (niebieski). Jest on zauważalnie szybszy.

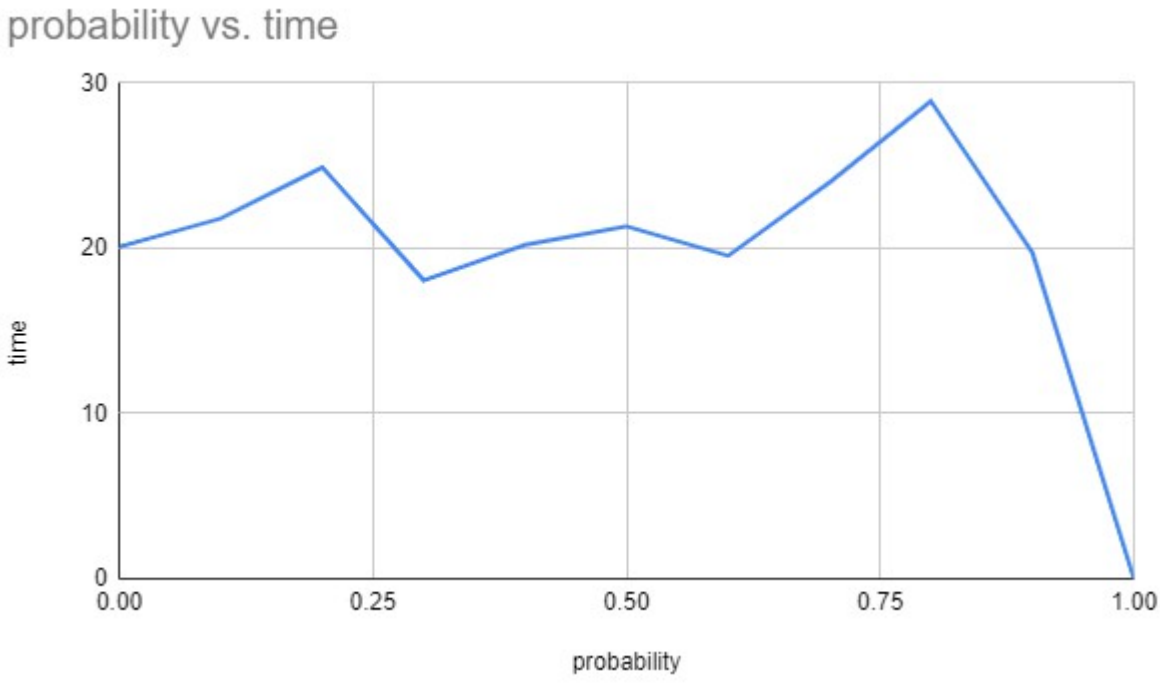
WPROWADZENIE PRAWDOPODOBIEŃSTWA

Dodaliśmy wymaganą opcję, polegającą na wprowadzeniu możliwości modyfikacji prawdopodobieństwa d takiego że dla każdej kulki prócz pierwszej jest prawdopodobieństwo d (d od

0 do 1), że wygenerowana kulka nie otrzyma koloru losowo, tylko będzie miała taki kolor jak poprzednia. W przeciwnym wypadku kolor losowany jest normalnie.

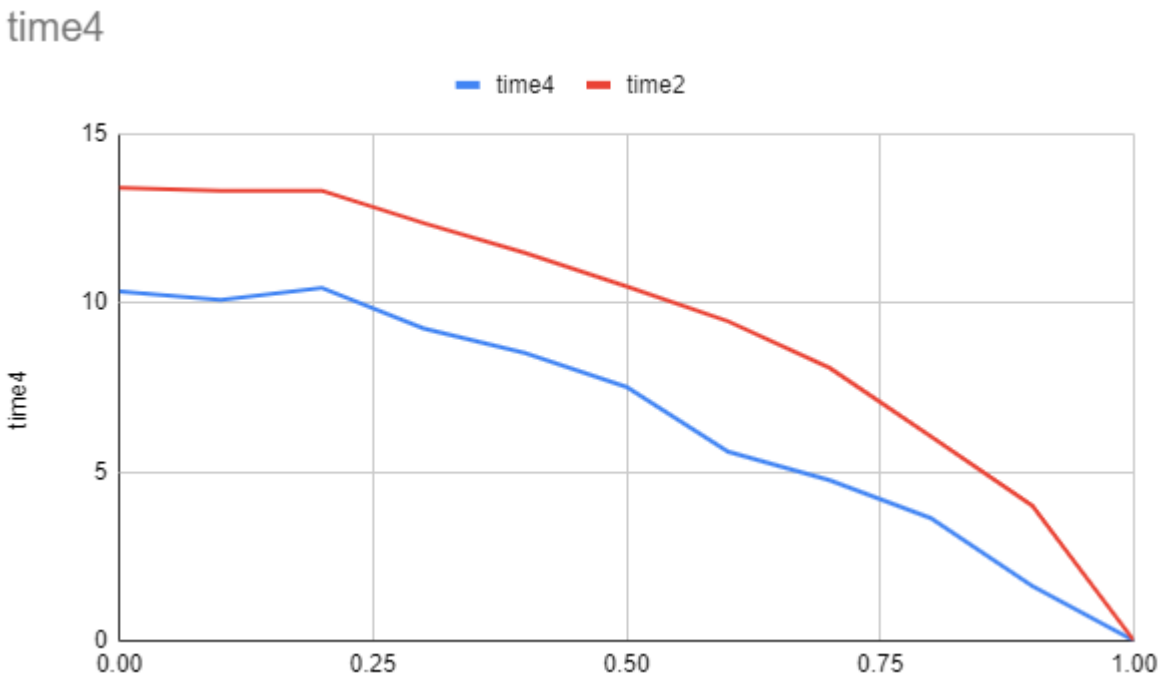
ALGORYTM 1 A PRAWDOPODOBIEŃSTWO

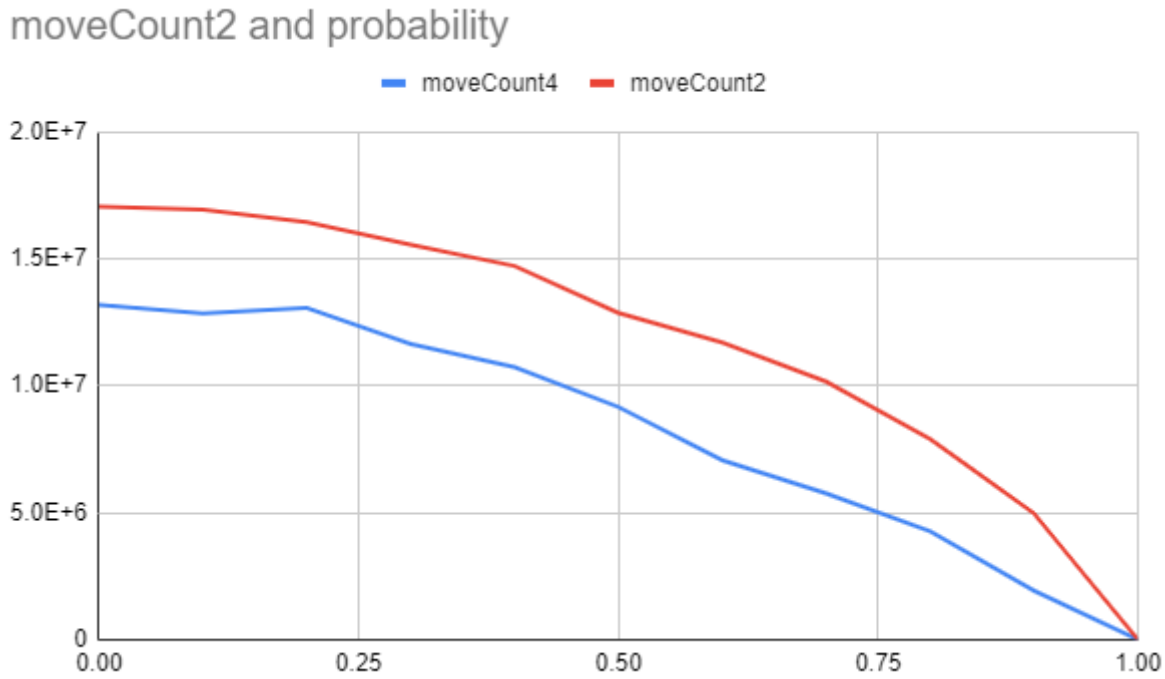




Wiemy, że algorytm 1 nie traktuje takich długich sekwencji specjalnie, więc nie dziwią wyniki oznaczające, że (oprócz oczywistej sytuacji d=1), nie ma wyraźnej korelacji między prawdopodobieństwem generowania długich ciągów, a prędkością sortowania.

ALGORYTM 2 I ALGORYTM 2 A PRAWDOPODOBIEŃSTWO



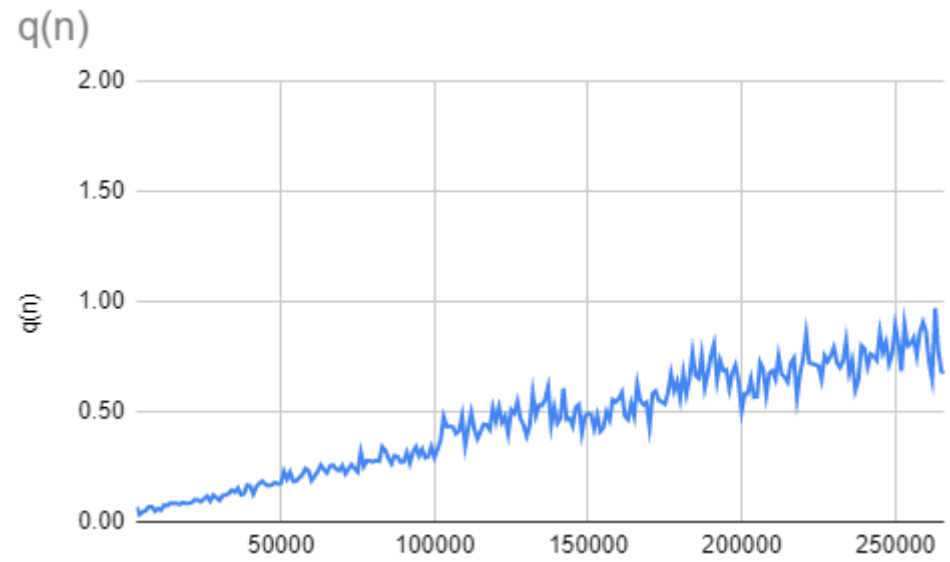


Jako że algorytm 4 priorytetowo traktuje długie sekwencje monokolorowe, występuje wyraźna korelacja między ilością długich ciągów monokolorowych, a prędkością rozwiązania. Algorytm 4 sprawdza się też lepiej od algorytmu 2, co zgadza się z teorią-w końcu z myślą o takich ciągach ten algorytm był tworzony.

WSPÓŁCZYNNIKI ZGODNOŚCI OCENY TEORETYCZNEJ Z POMIAREM CZASU

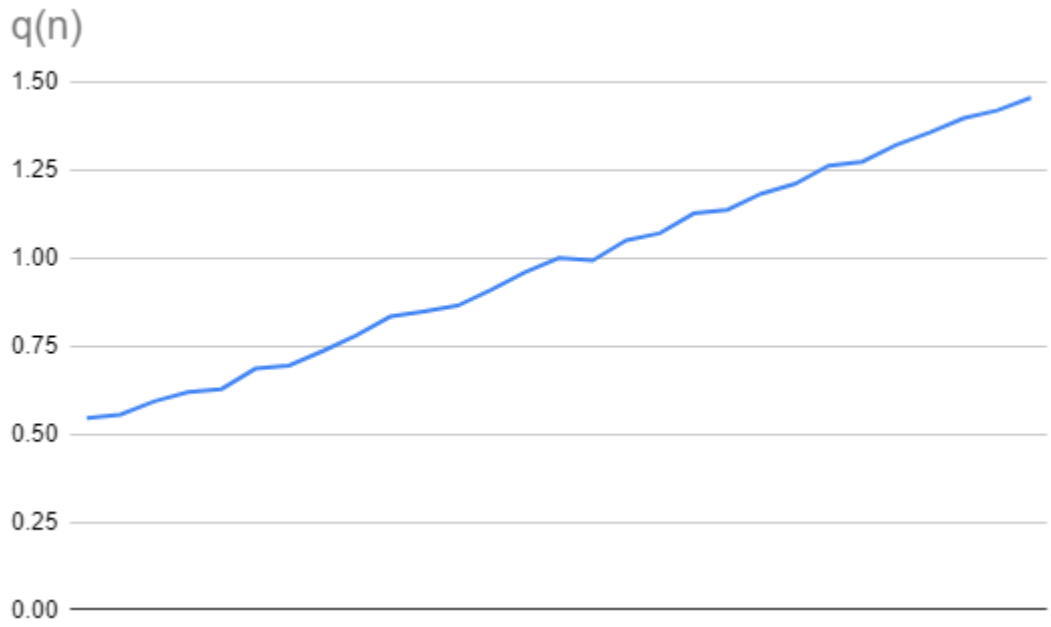
Zgodnie z poprzednimi wnioskami widzimy, że wskutek własności wektora w C++ złożoność będzie większa niż przewidywana. Z algorytmu teoretycznie (i praktycznie pod względem liczby ruchów) O(n) robi się O(n^2), a z O(n^2) - O(n^3). Poniżej zamieszczamy wykresy przy założeniu, że algorytm 1 powinien mieć złożoność liniową, a algorytm 2 kwadratową (tzn. bez wzięcia pod uwagę dodatkowego czynnika wprowadzonego przez wektory).

ALGORYTM 1



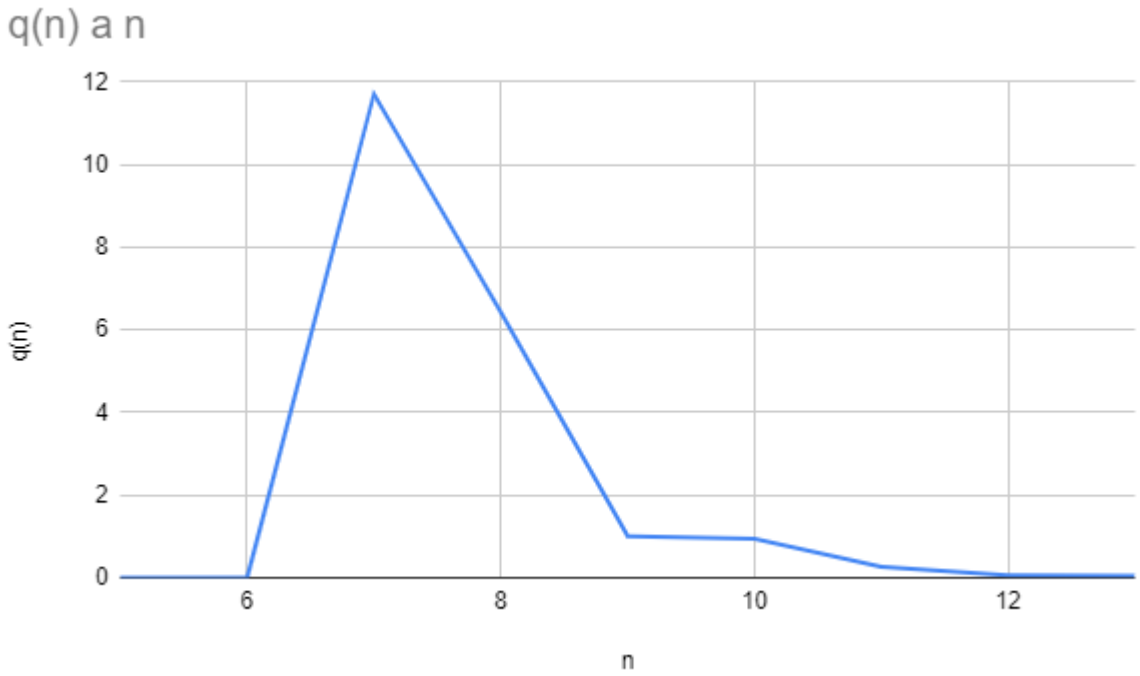
Zgodnie z przewidywaniami, funkcja jest rosnąca, czyli złożoność jest niedoszacowana.

ALGORYTM 2



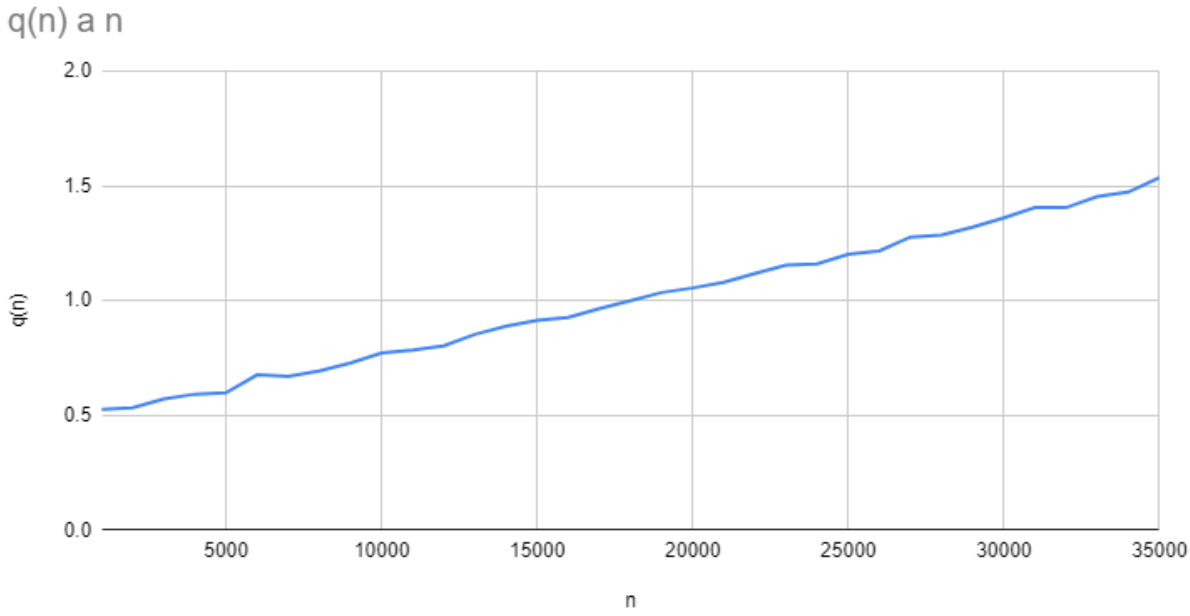
Tak jak w 2, funkcja jest rosnąca, czyli złożoność jest niedoszacowana.

ALGORYTM 3



Dla algorytmu 3 koncepcja dopasowania jest nierealizowalna praktycznie, nie wiadomo nawet jaką on miałby złożoność pesymistyczną-w tym rozwiązaniu mógłby nawet się zapętlać.

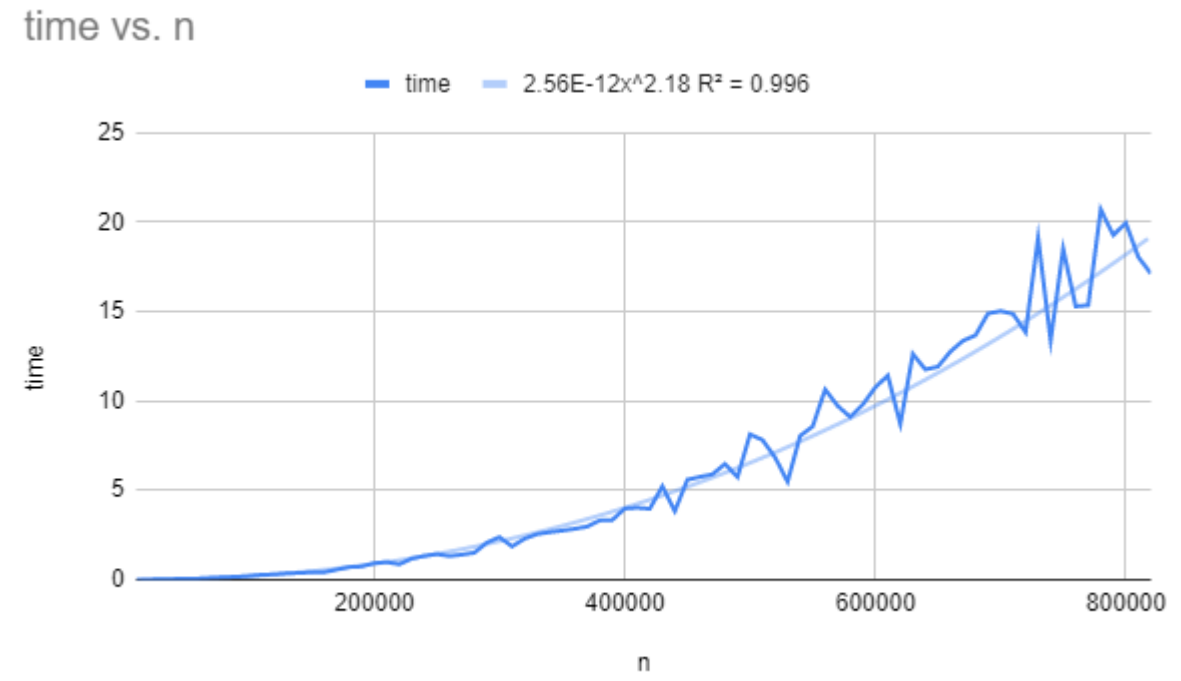
ALGORYTM 4

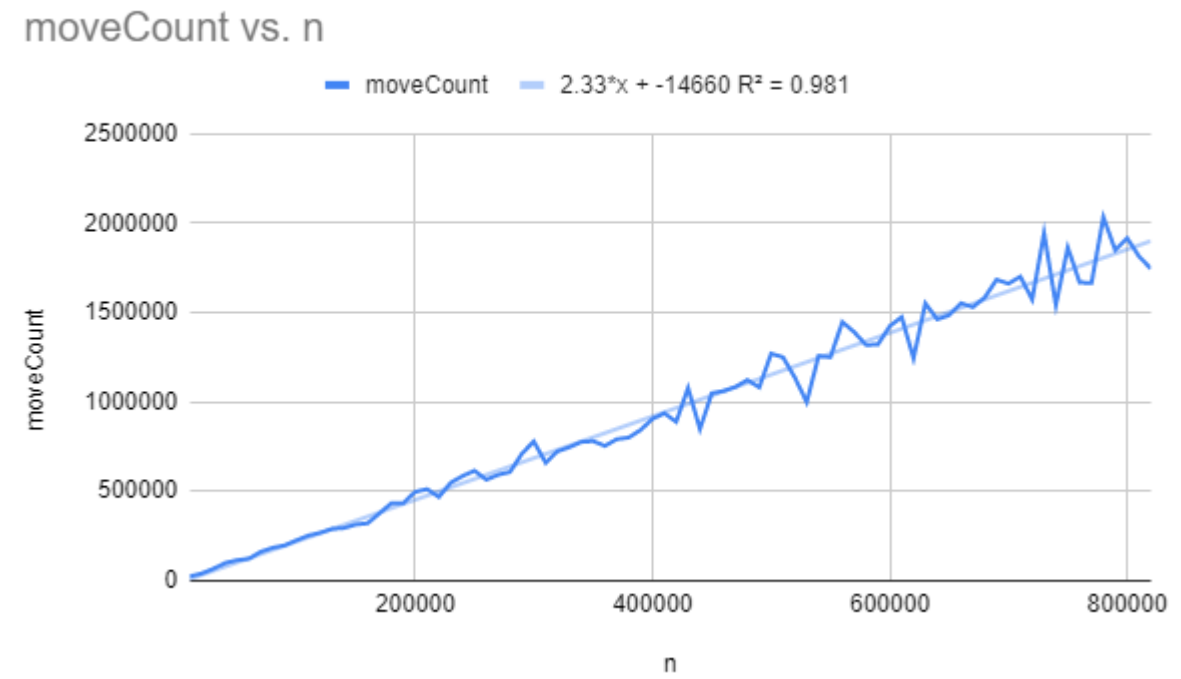


Tutaj również jest niedoszacowanie, analogicznie do algorytmu 2

DODATEK-maksymalnie zoptymalizowana wersja algorytmu 1

Dotychczasowe pomiary odbywały się bez żadnych flag optymalizujących. Dodatkowo zamieszczamy wyniki otrzymane dla algorytmu 1 po modyfikacji funkcji przestawiania kulek (co zauważalnie poprawiło prędkość, acz nie zmieniło klasy złożoności), a także dodaniu flagi kompilatora *-Ofast*





Tutaj próbkowaliśmy co 10000, a nie co 1000, więc oczywiście wyniki są mniej dokładne. Niemniej jednak widać wyraźnie nawet 3-krotną (patrząc na czas dla 800000 kulek) redukcję wymaganego czasu.