Warszawa 2.01.2019

Dokumentacja projektu na Analizę Algorytmów, semestr 18Z

**Autor: Adrian Nadulny, 283706**

Treść zadania 9:

Mamy zestaw S niełamalnych patyków o długości si, gdzie i∈(1,2,...,S). Zaproponuj algorytm wyliczający na ile sposobów można zbudować kwadrat przy użyciu 6 z tych patyków i wyznaczyć, których patyków należy użyć.

Zrozumienie problemu

Posiadamy tablicę S elementową liczb naturalnych mogących powtarzać się i należących do przedziału od 1 do S włącznie. Za każdym razem z tej tablicy jest wybieranych 6 różnych elementów, a następnie jest sprawdzany warunek dotyczący poprawności zbudowania kwadratu z wybranych liczb. Taki warunek może zajść dla 6 wybranych liczb tylko w dwóch wariantach:

* gdy największe dwie liczby są sobie równe, a pozostałe 4 można tak dobrać w pary, że po zsumowaniu par dostajemy równość każdej z sum z maksymalną liczbą w wybranym 6 elementowym podzbiorze.
* gdy największe trzy liczby są sobie równe, a po zsumowaniu pozostałych trzech liczb dostajemy równość sumy z największą liczbą w wybranym 6-elementowym podzbiorze.

Moim zadaniem jest sprawdzenie ile jest takich kombinacji 6-elementowych z których można zbudować kwadrat oraz wypisanie tych kombinacji do oddzielnego pliku. Dodatkowo z treści zadania wynika, że kolejność boków podczas budowania kwadratu nie jest istotna.

Algorytm naiwny (brute-force) Posiadający złożoność: O(n^6)

Algorytm opiera się na wykorzystaniu 6 głównych zagnieżdżonych pętli. Każda z pętli głównych jest odpowiedzialna za wybieranie jednego elementu dla podzbioru 6-elementowego. Z utworzonego podzbioru szukana jest liczba największa wraz ze zliczeniem ile razy pojawiła się ona w podzbiorze. Dodatkowo odbywa się też zliczanie sumy wszystkich wybranych 6-elementów . Następnie rozpatrywany jest jeden z dwóch warunków, które dotyczą odpowiednio wyżej wymienionych wariantów zbudowania kwadratu. Wyboru dokonuje się w zależności od sumy wszystkich elementów oraz od ilości wystąpienia największej liczby w podzbiorze. Poniżej przedstawiłem przebieg algorytmu.

Dane wejściowe:

* length – liczba niełamalnych patyków.
* algorithmTable[length] – tablica zawierająca liczby naturalne oznaczające długości kolejnych patyków.
* myFile – strumień ofstrem wskazujący miejsce do którego należy zapisywać dane wyjściowe.

Dane wyjściowe:

* Liczba kwadratów jakie można zbudować przy wejściowych danych.
* Powstały plik TXT zawierający 6-elementowe kombinacje liczb za pomocą których można zbudować kwadrat.

Przebieg algorytmu:

1. Tworzę za pomocą konstruktora obiekt *Algorithm*, który dostaje na starcie już wypełnioną tablicę S-elementową wraz z parametrem jej długość. Tablica ta zapamiętana jest od teraz pod nazwą *algorithmTable*, a jej długość oznaczona jest przez zmienną *length*.
2. Wpisuję do wskazanego pliku kolejne elementy zawarte w tablicy w celu późniejszej walidacji danych wyjściowych.
3. Jeżeli *length* *< 6* to oznacza, że zostały podane błędne dane wejściowe i następuje zakończenie algorytmu ze zwróceniem wyniku 0.
4. Tworzę pustą tablicę 6-elementową *table* i licznik *squaresCounter* *=* *0*, który zlicza te kombinacje 6-elementowe za pomocą których można zbudować kwadrat.
5. Korzystam z pętli głównej i pięciu zagnieżdżonych w niej pętli, żeby sprawdzić wszystkie możliwe warianty wyboru 6 liczb z tablicy *algorithmTable*. W każdej z pętli wybieram jeden z elementów pomocniczej tablicy *table*.
6. Tworzę pomocniczy licznik *counterLargestNumber* zliczający wystąpienia największej liczby w tablicy *table*, oraz zmienne *maxElement* i *sum* *=* *0*.
7. W pętli pomocniczej wyznaczam *maxElement* będący największą liczbą w tablicy *table*. Dodatkowo zliczam ile razy pojawił się *maxElement* w *table* oraz sumuję wszystkie 6 elementów z tablicy pomocniczej.
8. Jeżeli *counterLargestNumber* *==* *3* i *sum* *==* *maxElement* *\** *4* to oznacza, że można zbudować kwadrat za pomocą liczb znajdujących się w *table*. Na trzy boki kwadratu przypadają 3 liczby tej samej wielkości oraz czwarty bok jest zbudowany z sumowanych trzech pozostałych liczb.
9. Jeżeli jednak *counterLargestNumber* *==* *2* i *sum* *==* *maxElement* *\** *4* to sprawdzam jeszcze czy za pomocą odpowiedniej kombinacji liczb z *table* można zbudować kwadrat o dwóch bokach złożonych z pojedynczych liczb i o dwóch pozostałych bokach zbudowanych z par liczb. Zbudowanie kwadratu będzie możliwe tylko i wyłącznie wtedy, gdy znajdę jedną parę liczb, której suma jest równa *maxElement*. Mam wtedy pewność, że ze względu na warunek wejściowy, suma dwóch pozostałych liczby różnych od *maxElement* będzie równa *maxElement*.
10. Dla powyższych dwóch kroków, znalezioną kombinację liczb z których można zbudować kwadrat zapisuję do pliku TXT i zwiększam licznik *squaresCounter* o 1.
11. Jeżeli nie zostały jeszcze sprawdzone wszystkie kombinacje 6-elementowe z tablicy *algorithmTable* to następuje powrót do punktu 5, w przeciwnym zwracana jest liczba *squaresCounter* jako wynik pracy algorytmu.

Algorytm własnej heurystyki Posiadający złożoność: O(n^3)

Algorytm opiera się na wykonaniu na samym wstępie sortowania liczb malejąco dla tablicy *algorithmTable*. Następnie w pętli przeglądając tablicę od początku, czyli zaczynając od sprawdzania największej liczby, szukane są i zliczane kolejne powtarzające się liczby. Dodatkowo wykonywane są dwie funkcje dotyczące zliczania możliwych wariantów zbudowania kwadratu z 6 patyków wśród których co najmniej dwa największe mają długość liczby na której zatrzymaliśmy zliczanie jej powtórzeń w pętli głównej. Jedna z funkcji dotyczy zliczania takich kombinacji liczb dla których powstaje kwadrat o dwóch bokach zbudowanych z par liczb. Druga funkcja zaś zlicza takie kombinacje liczb dla których możliwe jest zbudowanie kwadratu o boku złożonym z trzech liczb i jest wykonywana dla liczby która w głównej pętli wystąpiła co najmniej trzy razy. Poniżej przedstawiłem przebieg algorytmu.

Dane wejściowe:

* length – liczba niełamalnych patyków.
* algorithmTable[length] – tablica zawierająca liczby naturalne oznaczające długości kolejnych patyków.
* myFile – strumień ofstrem wskazujący miejsce do którego należy zapisywać dane wyjściowe.

Dane wyjściowe:

* Liczba kwadratów jakie można zbudować przy wejściowych danych.
* Powstały plik TXT zawierający 6-elementowe kombinacje liczb za pomocą których można zbudować kwadrat.

Przebieg algorytmu:

1. Tworzę za pomocą konstruktora obiekt *Algorithm*, który dostaje na starcie już wypełnioną tablicę S-elementową wraz z parametrem jej długość. Tablica ta zapamiętana jest od teraz pod nazwą *algorithmTable*, a jej długość oznaczona jest przez zmienną *length*.
2. Za pomocą algorytmu *quicksort* sortuję malejąco liczby z tablicy *algorithmTable*.
3. Wpisuję do wskazanego pliku kolejne elementy zawarte w posortowanej tablicy w celu późniejszej walidacji danych wyjściowych.
4. Jeżeli *length* *<* *6* to oznacza, że zostały podane błędne dane wejściowe i następuje zakończenie algorytmu ze zwróceniem wyniku 0.
5. Tworzę licznik *counterRepeatingNumber*, który zlicza wystąpienia w *algorithmTable* tej samej liczby na kolejnych pozycjach oraz tworzę licznik *squaresCounter*, zliczający kombinacje liczb z których można zbudować kwadrat.
6. W pętli wyznaczam nowe wartości dla *counterRepeatingNumber*. Jeżeli skończę zliczanie powtarzających się po sobie liczb i *counterRepeatingNumber* będzie większy od 1 to zwiększam licznik *squaresCounter* o zwrócone wartości z funkcji pomocniczych *searchingTwoSidesOfSquare* oraz *searchingThreeSticks*, gdzie obydwie funkcje zwracają nowe kombinacje zbudowania kwadratu z 6 liczb. Obydwie funkcje jako dane wejściowe przyjmują trzy parametry: *position* – liczba naturalna oznaczająca indeks od którego funkcja zaczyna przeszukiwanie w tablicy *algorithmTable*; *counterRepeatingNumber* – jest opisany powyżej; *myFile* – strumień ofstrem wskazujący miejsce do którego należy zapisywać dane wyjściowe.
7. Funkcja *searchingTwoSidesOfSquare*:

Dla zbioru 6-elementowego mam już wybrane 2 największe liczby o tej samej wartości i poszukuję dwóch takich par liczb, których sumy są równe wartości największej liczby w tym zbiorze. Funkcja jest opisana poniżej:

* 1. Tworzę *vector* o nazwie *combination*, licznik *squaresCounter = 0*, tymczasowe liczniki pomocnicze *tmpi*, *tmpj*, *tmpNumberOfCombination*.
  2. W pętli głównej za pomocą dwóch pomocniczych zmiennych lokalnych  *i*  oraz *j*  przeszukuję część tablicy *algorithmTable* od indeksu *position* do ostatniego indeksu tablicy. Przeszukiwanie odbywa się jednocześnie w dwóch kierunkach, gdzie z każdej ze stron zbliżam się do elementu równego połowie wartości liczby, którą zliczałem przed wejściem do tej funkcji.
  3. Jeśli suma liczb znajdujących się w *algorithmTable* pod indeksami *i*  oraz *j* będzie większa od wartości liczby, która określa długość boku budowanego kwadratu to oznacza, że trzeba znaleźć liczby dające mniejszą sumę więc inkrementuję *i* .
  4. Jeżeli jednak ta suma byłaby mniejsza to wtedy poszukuję dwóch liczb dających większą sumę, dlatego dekrementuję *j* . W przeciwnym przypadku zliczam powtórzenia wartości liczb pod indeksami *i* oraz *j* , zwiększając odpowiednio *tmpi* oraz *tmpj*.
  5. Wyznaczam *tmpNumberOfCombination* jako liczbę wszystkich kombinacji wyboru po jednej z wartości *algorithmTable[i]* i *algorithmTable[j]* ze zliczonych powtórzeń tmpi oraz tmpj.
  6. W pętli pomocniczej tak długo jeżeli mogę to pobieram z tablicy *combination* trzy kolejne elementy, które odpowiednio oznaczam jako *n* , *a* , *b* , gdzie *n*  to liczba wystąpień tej samej kombinacji dla liczb *a*  oraz *b* .
  7. Zwiększam *squaresCounter* o wyznaczoną liczbę wariantów zbudowania dwóch boków kwadratu z pary *a*  i *b* oraz pary liczb znajdujących się pod indeksami *i* oraz *j* w *algorithmTable*.
  8. Zapisuję w pliku TXT powtarzające się warianty 6-elementów za pomocą których można zbudować kwadrat.
  9. Jeśli można zbudować kwadrat z dwóch par liczb o wartościach wskazywanych przez indeksy *i* oraz *j* to zliczam liczbę takich kombinacji i zwiększam odpowiednio *squaresCounter* oraz zapisuję te kombinacje w pliku TXT.
  10. Do tablicy *combination* dodaje odpowiednio trzy nowe wartości *tmpNumberOfCombination*, *algorithmTable[i]*, *algorithmTable[j]*.
  11. Wracam do pętli głównej chyba, że jest spełniony warunek wyjścia z pętli to zwracam liczbę *squaresCounter*.

1. Funkcja *searchingThreeSticks*:

Dla zbioru 6-elementowego mam już wybrane 3 największe liczby o tej samej wartości i poszukuję trzech liczb, których suma jest równa wartości największej liczby w zbiorze. Funkcja jest opisana poniżej:

* 1. Jeśli nie zliczono trzech powtórzeń wartości liczby mającej stanowić największą liczbę w podzbiorze 6-elementowym to dalsze kroki algorytmu nie mogą zostać wykonane. Zwracana jest liczba 0.
  2. Tworzę licznik *squaresCounter* *=* *0*, tymczasowe liczniki pomocnicze *tmpi*, *tmpj*, *tmpk*.
  3. Za pomocą indeksów *i* , *j* oraz *k* określam z jakich liczb będę budował ostatni, czwarty bok kwadratu.
  4. W pętli głównej wybieram pierwszą z trzech poszukiwanych liczb. Określam tą liczbę poprzez indeks *i* oraz zliczam ile razy powtarza się ona w *algorithmTable*.
  5. Dla szukanych dwóch pozostałych liczb wyznaczam zmienną searchingSum będącą wartością sumy poszukiwanych liczb.
  6. W pętli pomocniczej podobnie jak to było w funkcji *searchingTwoSidesOfSquare* przeszukuję jednocześnie w dwóch kierunkach część tablicy *algorithmTable*. Ustawiam odpowiednio indeksy *j* (jako indeks końca tablicy) oraz *k* (jako indeks równy *i* jeżeli zliczono powtórzenia kolejnych liczb w pętli głównej lub jako indeks równy *i+1)*. Podczas przeszukiwaniaz każdej ze stronzbliżam się doelementu równego połowie wartości poszukiwanej sumy.
  7. Jeśli suma liczb znajdujących się w *algorithmTable* pod indeksami *j*  oraz *k* będzie większa od wartości liczby *searchingSum* to oznacza, że trzeba znaleźć liczby dające mniejszą sumę więc inkrementuję *k* .
  8. Jeżeli jednak ta suma byłaby mniejsza to wtedy poszukuję dwóch liczb dających większą sumę, dlatego dekrementuję *j* . W przeciwnym przypadku zliczam powtórzenia wartości liczb pod indeksami *k* oraz *j* , zwiększając odpowiednio *tmpk* oraz *tmpj*.
  9. Sprawdzam cztery warianty w których można zbudować kwadrat o jednym boku składającym się z sumy trzech liczb. Te warianty to:
* Jeśli indeksy  *i* , *j* oraz *k*  są sobie równe i znaleziono co najmniej 3 powtarzające się liczby o wartościach *algorithmTable[i]*  ,to wtedy bok kwadratu zbudowany będzie z trzech liczb o tej samej wartości.
* Jeśli tylko indeksy *i*  oraz *k* są sobie równe to bok kwadratu będzie zbudowany z sumy dwóch liczb o tej samej wartości i liczby mniejszej od pozostałych dwóch.
* Jeśli tylko indeksy *j*  oraz *k* są sobie równe to bok kwadratu będzie zbudowany z sumy dwóch liczb o tej samej wartości i liczby większej od pozostałych dwóch.
* Jeśli każdy z indeksów *i* , *j* oraz *k*  będzie różny od pozostałych to wtedy bok kwadratu będzie zbudowany z sumy trzech liczb o różnych wartościach.

Po spełnieniu warunków jednego z powyższych wariantów, wyznaczam liczbę znalezionych nowych kombinacji zbudowania kwadratu z 6-elementów i zwiększam *squaresCounter* o tą liczbę*.*

* 1. Zapisuję w pliku TXT powtarzające się warianty 6-elementów za pomocą których można zbudować kwadrat.
  2. Wracam do pętli pomocniczej lub do pętli głównej funkcji, jeżeli w pętli głównej nie sprawdziłem jeszcze wszystkich kombinacji wyboru trzech liczby. W przeciwnym przypadku zwracam liczbę *squaresCounter.*

1. Wyznaczam nową wartość dla *counterRepeatingNumber = 1*.
2. Jeżeli w głównej pętli nie doszliśmy do trzeciej liczby z końca tablicy *algorithmTable* to następuje powrót do punktu 6, w przeciwnym wypadku zwracana jest liczba *squaresCounter* jako wynik pracy algorytmu.

Podsumowanie

Algorytm własnej heurystyki opiera się o wstępne posortowanie malejąco tablicy wejściowej za pomocą algorytmu *quicksort* o złożoności nlog(n). Po posortowaniu elementów z tablicy wystarczy teraz zawężać poszukiwania kombinacji 6-elementowych do coraz mniejszych podzbiorów tablicy *algorithmTable*. Odbywa się to przez sprawdzanie od największego elementu w tablicy czy kolejne liczby z algorithmTable mają tą samą wartość. Jeżeli znaleziono powtarzające się wartości to dopiero teraz korzystam z dwóch funkcji pomocniczych, gdyż istnieje możliwość zbudowania kwadratu. W najgorszym przypadku kiedy kolejne trójki liczb mają tą samą wartość, wtedy za każdym razem wykonają się dwie pomocnicze funkcje *searchingTwoSidesOfSquare* o złożoności n^2 oraz *searchingThreeSticks* o największej złożoności n^3.

Zaprezentowany algorytm jest znacznie szybszy niż algorytm naiwny dla *n > 20*, jednakże czasy pracy algorytmu zaprezentowanej heurystyki dla tego samego *n* , mogą różnić się znacząco w zależności od liczb zapisanych w *algorithmTable*. Dzieje się tak ze względu na konieczność wykonania większej ilości obliczeń.

Budowa programu

Program będący rozwiązaniem problemu algorytmicznego został napisany w języku C++. Składa się z pliku głównego i dwóch plików nagłówkowych. Szczegółowy opis poszczególnych plików przedstawiłem poniżej.

Główny plik – main.cpp

Jest to plik w którym znajduje się interfejs użytkownika uruchamiany za pomocą funkcji UI(). Użytkownik za pomocą wpisania odpowiedniego numeru od 1 do 7 może wywołać funkcje główne z których każda przyjmuje na wejściu obiekt klasy *Sticks*. Te funkcje to:

* changeSticksNumber – zmienia wartość S.
* generateStickTableElements – losowe generowanie wartości w tablicy *stickTable*.
* generateOwnStickTableElements – użytkownik wprowadza nowe wartości do tablicy *stickTable*.
* viewAlgorithmsTimeMeasure – dokonywany jest pomiar czasu działania algorytmu naiwnego i algorytmu ulepszonego dla obecnego obiektu klasy *Sticks*.
* showStickTableElements – wypisywane są na ekranie obecne elementy zawarte w tablicy *stickTable*.
* generateTimeMeasureResults – w zależności od podanej przez użytkownika liczby naturalnej dokonywanych jest tyle głównych pomiarów czasów działania algorytmu naiwnego i ulepszonego. Użytkownik wprowadza też wielkość S dla każdego głównego pomiaru, który będzie składał się z 5 pomniejszych pomiarów czasu działania algorytmów przy tym samym rozmiarze tablicy *algorithmTable*, jednakże wartości w *algorithmTable* są dla każdego pomiaru losowo generowane.

W pliku tym znajdują się też funkcje pomocnicze odpowiedzialne za:

* sleep(int seconds) – wstrzymanie działania programu przez kilka sekund.
* getNumber() – wybór liczby głównych pomiarów czasowych dla algorytmów.
* getStickNumber() – wybór nowego S, czyli rozmiaru tablicy *stickTable*.
* pressKeyToGoBack() – powrót do głównego panelu opcji.
* optionScreen(Sticks &sticks) – wyświetlenie listy dostępnych opcji działania programu.
* algorithmsTimeMeasure(Algorithm &algorithm, string fileName) – wspomaga dokonanie jednego pomiaru czasowego dla algorytmu naiwnego oraz ulepszonego.
* algorithmsTimeMeasure(Algorithm &algorithm, string fileName, int p) – wspomaga dokonanie wielu pomiarów czasowych dla algorytmu naiwnego oraz ulepszonego, gdzie zmienna p determinuje końcowy napis pliku TXT w którym są zapisywane dane wyjściowe.
* saveTimeMeasureData(ofstream &myFile, int stickNumberTab[], double algorithmTimes[][5], int length) – wypisuje do pliku CSV zebrane czasy działania algorytmów wraz z obliczeniem średniej czasowej dla każdego pomiaru głównego. Dodatkowo w pliku CSV zapisywane są wartości zmienne S, oznaczające długości tablicy algorithmTable dla kolejnych serii pomiarów.

Algorithm.h

Plik zawierający klasę Algorithm, która zawiera implementację algorytmu naiwnego (funkcja algorithmNaive(ofstream &myFile)) oraz algorytm własnej heurystyki (funkcja algorithmMyHeuristic(ofstream &myFile)). Są tu też funkcje pomocnicze np.:

\* quicksort(int table[], int p, int r) – do sortowania elementów w tablicy;

\* newton(int n, int k) – do wyznaczania wartości Symbolu Newtona z n nad k;

\* searchingTwoSidesOfSquare(int position, int counterRepeatingNumber, ofstream &myFile) – do wyznaczania kombinacji zbudowania kwadratu, którego dwa boki będą zbudowane z dwóch par liczb znajdujących się w części tablicy algorithmTable między indeksami position i length-1;

\* searchingThreeSticks (int position, int counterRepeatingNumber, ofstream &myFile) – do wyznaczania kombinacji zbudowania kwadratu, którego jeden bok będzie zbudowany z trzech liczb znajdujących się w części tablicy algorithmTable między indeksami position i length-1.

Sticks.h

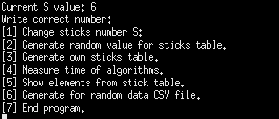
Plik nagłówkowy w którym zdefiniowana jest klasa *Sticks* przechowująca *stickNumber*, czyli liczbę patyków oraz tablicę *stickTable* w której zapisane są wartości długości kolejnych patyków. Dla obiektu klasy *Sticks* można zmieniać, czy też zwracać wartość *stickNumber* lub tablicę *stickTable*.

Wywoływanie programu

Aby wywołać program z poziomu konsoli, należy podać jego nazwę: ./aal .

Tryby pracy programu

Program umożliwia pracę w kilku trybach, jednakże najpierw użytkownik musi podać liczbę naturalną S większą od 5. Na ekranie wyświetli się wtedy obecna wartość S oraz lista możliwych opcje do wyboru.



- tryb 1

Po wpisaniu liczby 1, użytkownik proszony jest o podanie nowego S co zmienia długość tablicy *algorithmTable*.

- tryb 2

Po wpisaniu liczby 2, generowane są nowe wartości dla tablicy *algorithmTable*.

- tryb 3

Po wpisaniu liczby 3, użytkownik generuje nowe wartości dla tablicy *algorithmTable*. Wartości podawane przez użytkownika nie mogą być większe niż rozmiar tablicy, czyli zmienna S. Na bieżąco wyświetlane są wpisane już nowe wartości i komunikat ile jeszcze nowych wartości należy podać.

- tryb 4

Po wpisaniu liczby 4, mierzony jest czas działania algorytmu naiwnego i ulepszonego dla obecnego obiektu klasy Sticks. Na ekranie wypisane zostaną nazwy poszczególnych algorytmów wraz z czasami działania oraz liczbą możliwości zbudowania kwadratu jakie znalazł algorytm.

- tryb 5

Po wpisaniu liczby 5, wyświetlana jest na ekranie obecna zawartość tablicy *stickTable.*

- tryb 6

Po wpisaniu liczby 6 i podaniu odpowiednich danych wejściowych, generowany jest plik CSV, który zawiera czasy kolejnych pomiarów działania algorytmów, średnie czasowe oraz wartości S dla każdego pomiaru głównego.

- tryb 7

Po wpisaniu liczby 7, następuje zamknięcie programu.

Wyniki pomiarów czasu wykonywania algorytmów

Program został uruchomiony za pomocą komendy ./aal . Następnie po pojawieniu się zapytania dotyczącego wstępnej liczby patyków, podana została liczba naturalna większa od 5. Z okna dialogowego wybrano opcję 6 dotyczącą generowania losowych danych i zapisywania czasów pomiaru w pliku CSV, z którego dane można łatwo otworzyć w Excelu. Kolejnym krokiem było podanie liczby pomiarów, które chcemy wykonać. Dla zadanej liczby pomiarów trzeba było podać kolejne liczby S, których pomiar czasu miał dotyczyć. Następnie program dla każdej wprowadzonej przez użytkownika liczby dokonywał 5 niezależnych pomiarów, gdzie przy każdym pomiarze czasu generowane były nowe liczby dla tablicy S-elementowej . Po dłuższej lub krótszej chwili, w zależności od wprowadzonych danych, pomiary czasu zostają zakończone.

Algorytm naiwny

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Liczba S | Czasy dla poszczególnych wywołań programu | | | | | Średnia wszystkich wykonań |
| 10 | 0.000032103 | 0.000027473 | 0.000203201 | 0.000174112 | 0.000537708 | 0.0001949194 |
| 20 | 0.000949562 | 0.000877453 | 0.001093993 | 0.001103476 | 0.001085948 | 0.0010220864 |
| 30 | 0.018992334 | 0.01248891 | 0.017210263 | 0.012595668 | 0.012373439 | 0.014732123 |
| 40 | 0.07719526 | 0.077003911 | 0.079406255 | 0.077274263 | 0.075471042 | 0.077270146 |
| 50 | 0.29643367 | 0.30036019 | 0.29473573 | 0.29377926 | 0.29316186 | 0.29569414 |
| 55 | 0.58953468 | 0.57454533 | 0.56945847 | 0.55943801 | 0.56673805 | 0.57194291 |
| 60 | 0.999694 | 0.96118898 | 0.98726426 | 1.051499 | 0.9267387 | 0.98527698 |
| 65 | 1.5542435 | 1.4969565 | 1.5510607 | 1.6034003 | 1.6561686 | 1.5723659 |
| 70 | 2.3852417 | 2.4601232 | 2.5250384 | 2.4543848 | 2.5392028 | 2.4727982 |
| 75 | 3.829555 | 3.6737649 | 3.7265057 | 3.6792928 | 3.6621115 | 3.714246 |
| 80 | 5.4040381 | 5.7905238 | 5.5853739 | 5.9209384 | 5.7788148 | 5.6959378 |
| 85 | 8.1936751 | 7.948868 | 8.1162412 | 8.1977727 | 8.0803685 | 8.1073851 |
| 90 | 11.203722 | 10.992328 | 11.539786 | 11.234091 | 11.98466 | 11.390917 |
| 95 | 16.563034 | 16.012008 | 16.522265 | 16.226673 | 16.165582 | 16.297912 |
| 100 | 21.269272 | 21.630634 | 25.377084 | 22.910563 | 21.269073 | 22.491325 |

|  |  |  |
| --- | --- | --- |
| Algorytm z asymptotyką O(T(n)) | | |
| n | t(n)[ms] | q(n) |
| 10 | 0,194919 | 9,34935368 |
| 20 | 1,022086 | 0,76600951 |
| 30 | 14,73212 | 0,96931366 |
| 40 | 77,27015 | 0,90485358 |
| 50 | 295,6941 | 0,90771438 |
| 55 | 571,9429 | 0,99106712 |
| 60 | 985,277 | 1,01292516 |
| 65 | 1572,366 | 1 |
| 70 | 2472,798 | 1,00815419 |
| 75 | 3714,246 | 1,0009897 |
| 80 | 5695,938 | 1,04220185 |
| 85 | 8107,385 | 1,03108334 |
| 90 | 11390,92 | 1,02808768 |
| 95 | 16297,91 | 1,06344947 |
| 100 | 22491,33 | 1,07880156 |

Algorytm własnej heurystyki

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Liczba S | Czasy dla poszczególnych wywołań programu | | | | | Średnia wszystkich wykonań |
| 10 | 1.7717e-05 | 1.6251e-05 | 0.000197293 | 0.000162934 | 0.00131895 | 0.000342629 |
| 20 | 3.1722e-05 | 0.001869532 | 0.003198123 | 0.000209808 | 0.00118976 | 0.001299789 |
| 30 | 7.43e-05 | 0.000258976 | 0.000234906 | 0.00022237 | 0.000306861 | 0.0002194826 |
| 40 | 0.000199325 | 0.000365508 | 0.000389599 | 0.000393297 | 0.000392383 | 0.0003480224 |
| 50 | 0.000449855 | 0.000653913 | 0.00060538 | 0.000569149 | 0.000540761 | 0.0005638116 |
| 55 | 0.000509349 | 0.000751525 | 0.000631771 | 0.000796741 | 0.000764944 | 0.000690866 |
| 60 | 0.000889933 | 0.000793397 | 0.00087143 | 0.000919752 | 0.001066246 | 0.0009081516 |
| 65 | 0.001151216 | 0.001271414 | 0.002158833 | 0.001352418 | 0.001360543 | 0.0014588848 |
| 70 | 0.001387568 | 0.001885356 | 0.001311242 | 0.001473078 | 0.001042513 | 0.0014199514 |
| 75 | 0.001541596 | 0.001470898 | 0.002116532 | 0.001359819 | 0.001258748 | 0.0015495186 |
| 80 | 0.002188242 | 0.002858737 | 0.002721402 | 0.001444113 | 0.002611966 | 0.002364892 |
| 85 | 0.001734766 | 0.002395885 | 0.001995795 | 0.002221516 | 0.003189814 | 0.0023075552 |
| 90 | 0.003221389 | 0.003325839 | 0.005151 | 0.00322339 | 0.002748915 | 0.0035341066 |
| 95 | 0.003773165 | 0.004453892 | 0.003709295 | 0.002244634 | 0.003818088 | 0.0035998148 |
| 100 | 0.003303153 | 0.003636902 | 0.004305732 | 0.004083463 | 0.004442855 | 0.003954421 |

|  |  |  |
| --- | --- | --- |
| Algorytm z asymptotyką O(T(n)) | | |
| n | t(n)[ms] | q(n) |
| 10 | 0,342629 | 81,49285207 |
| 20 | 1,299789 | 38,64366148 |
| 30 | 0,2194826 | 1,933444592 |
| 40 | 0,3480224 | 1,293369521 |
| 50 | 0,5638116 | 1,072801551 |
| 55 | 0,690866 | 0,987645639 |
| 60 | 0,9081516 | 1 |
| 65 | 1,4588848 | 1,263503148 |
| 70 | 1,4199514 | 0,984633843 |
| 75 | 1,5495186 | 0,873591505 |
| 80 | 2,364892 | 1,098592804 |
| 85 | 2,3075552 | 0,893697865 |
| 90 | 3,5341066 | 1,153048342 |
| 95 | 3,5998148 | 0,998630304 |
| 100 | 3,954421 | 0,940542235 |