



Dokumentacja

Analiza Algorytmów - Projekt

Opracował – Paweł Walczak

Prowadził – dr inż. Tomasz Trzciński

SPIS TREŚCI:

- 1. Treść zadanie.**
- 2. Użyte narzędzia i technologie.**
- 3. O rozwiązaniu zadania.**
- 4. Analiza pokrycia przypadków.**
- 5. Opis metody rozwiązania.**
- 6. Słów kilka o wypisaniu kombinacji**
- 7. Testowanie**
- 8. GUI**
- 9. Analiza algorytmu teoretyczna**
- 10. Analiza złożoności algorytmu (pomiar czasu).**
- 11. Listing klas i plików z objaśnieniami**
- 12. Github**

1. Treść zadania.

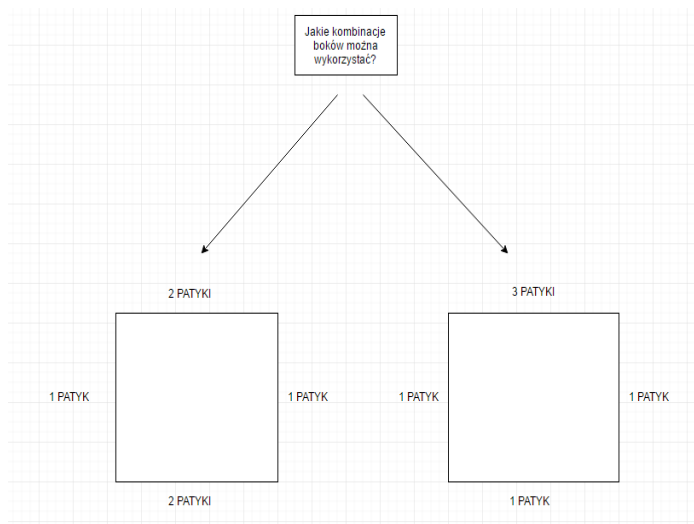
Mamy zestaw S niełamałych patyków o długości s_i , $i \in (1, 2, \dots, S)$. Zaproponuj algorytm wyliczający na ile sposobów można zbudować kwadrat przy użyciu 6 z tych patyków i wyznaczy, które patyki należy użyć.

2. Użyte narzędzia oraz technologie.

Wybrany przeze mnie język programowania to **C++**, który znajduje zastosowanie w problemach, w których istotna jest efektywność i szybkość działania (co de facto jest wymaganiem w problemach optymalizacji). Korzystałem z IDE **CLion** oraz kompilatora **MinGW**. Zastosowałem biblioteki **gtest** (opracowane przez Google), w celu opracowywania projektu zgodnie z założeniami TDD oraz testowania poprawności działania algorytmu.

3. O rozwiązaniu zadania.

Korzystając z 6 patyków kwadrat można zbudować na dwa sposoby:



Rozpoczęcie projektu rozpocząłem od napisania kodu dla rozwiązania typu **Bruteforce**. Rozwiązanie tego typu jest dość oczywiste i nie przejmujemy się w nim złożonością algorytmu. W przypadku mojego projektu, wystarczyło dla każdej 6-elementowej permutacji danego zbioru patyków sprawdzić czy da się z niej zbudować kwadrat. Rozwiązanie tego typu ma złożoność $O(n^6)$. W programie zawarłem opcję uruchomienia algorytmu typu Bruteforce. Nie będę opisywał tutaj dokładnej implementacji algorytmu, gdyż nie jest to celem dokumentacji.

Lepsze rozwiązanie...

Rozwiązanie bardziej optymalne jest już cięższe do znalezienia oraz implementacji.

Pierwszym krokiem w moim rozwiązaniu jest posortowanie zbioru. Mogłem skorzystać z wbudowanych bibliotek lecz postanowiłem zaimplementować samodzielnie algorytm sortowania **QuickSort**. Owy algorytm posiada złożoność $O(n \log n)$. Należy podkreślić, że algorytm ten może działać szybciej, zależy to od tego jak wygląda zbiór wejściowy (czy rzeczywiście kolejność elementów jest losowa).

Rozszerzyłem funkcjonalność algorytmu QuickSort o utworzenie tablicy której elementami będą indeksy patyków z oryginalnego zbioru (tablicy), porządkowane po wykonaniu sortowania:

intoriginalInput [] = {5,1,3} , po wykonaniu QuickSort

intsortedArray[] = {1,3,5};

intarrayOfIndices[] = {2,3,1}

Tablicę indeksów wykorzystam w algorytmie BruteForce do przedstawienia których patyków należy użyć do utworzenia kwadratu.

Drugim krokiem jest wyznaczenie ilości kombinacji patyków, z których możemy utworzyć kwadrat. Przydatna w implementacji algorytmu okaże się **Analiza Pokrycia Przypadków**:

ANALIZA POKRYCIA PRZYPADKÓW.

W zadaniu należy rozpatrzyć różne przypadki danych wejściowych z jakimi możemy się spotkać i z jakimi radzić sobie musi algorytm.

1. Jeżeli dane wejściowe liczą mniej niż 6 patyków to przerwać działania programu, gdyż nie znajdziemy rozwiązania.
2. Jeżeli dane wejściowe liczą 6 patyków to:
 - a. Sprawdzić czy istnieją duplikaty
 - i. Jeśli tak to należy sprawdzić czy da się z nich złożyć trójkąt (kombinacja boków (2,2,1,1) oraz (3,1,1,1))
 - ii. Jeśli nie to nie da się złożyć kwadratu (przerwij program , zwróć 0).
3. Gdy dane wejściowe liczą powyżej 6 patyków, sytuacja zaczyna być bardziej skomplikowana i należy rozpatrzyć kilka przypadków jakie musi pokryć algorytm.
 - a. Szukamy kombinacji (3,1,1,1):
 - i. Patyków o takich samych długościach mamy liczbę n ($n \geq 3$) (mowa o tych które pojedynczo tworzą bok) .
POKRYCIE:
Trzy z nich (dla potrzeby realizacji zadania) możemy wybrać na $\binom{n}{3}$ sposobów.
 - ii. Wszystkie patyki (z boku z 3 patyków) mają różne długości.
POKRYCIE:
Przechowuj tablicę zawierającą ilość wystąpień par patyków sumujących się do danych długości i aktualizuj w raz z każdą inkrementacją w pętli (która przeszukuje
 - iii. Dwa patyki mają tę samą długość , a trzeci jest większy/mniejszy.
POKRYCIE:
Do pokazania na kodzie...

- iv. Wszystkie patyki mają tę samą długość.
POKRYCIE:
Do pokazania na kodzie...
- b. Szukamy kombinacji (2,2,1,1):
 - i. Patekóv pojedynczo tworzących boki jest n ($n \geq 2$).
POKRYCIE:
Dwa z nich (dla potrzeby realizacji zadania) możemy wybrać na $\binom{n}{2}$ sposobów.
 - ii. Wszystkie patyki z boków 2,2 są tej samej długości i dowolna para sumuje się do długości patyka który tworzy jeden bok.
POKRYCIE:
Z dwumianu newtona policz kombinacje $\binom{n}{4}$, gdzie n to liczba danych patyków o tej samej długości.
 - iii. Dwa patyki które stworzą jeden bok, są tej samej długości.
POKRYCIE:
Z dwumianu newtona policz kombinacje $\binom{n}{2}$, gdzie n to liczba danych patyków o tej samej długości.
 - iv. Będą po 2 patyki tej samej długości czyli stworzą boki (A+B),(A+B), gdzie A+B sumuje się do wartości patyka, który tworzy pojedynczy bok:
POKRYCIE:
Policz wartości dwóch kombinacji newtona : $\binom{a}{2}$, gdzie a to liczba patyków A, oraz $\binom{b}{2}$, gdzie b to liczba danych patyków B i przemnoż przez siebie.
 - v. Wszystkie patyki tworzące boki (2,2) są różnej długości.
POKRYCIE:
Do wyjaśnienia na kodzie...

Opis metody rozwiązania:

Najpierw zajmijmy się opisem algorytmu wyznaczania kombinacji gdzie występują 3 patyki o tej samej długości, a kolejne trzy sumują się do długości tripletu. Tworzymy tablicę, której wartościami są liczby wystąpień patyków od długości danej indeksem.

```
for (inti=1;i<=n;i++) array[i] = tab[i-1], onesArray[array[i]]++;
```

Następnie iterujemy po posortowanym uprzednio zbiorze patyków. Obliczamy działania:

```
for (inti=1;i<=n;i++){
for (ints=i+1;s<=n;s++)
if (array[s]>array[s-1]) {
ans += 1ll * onesArray[array[s]] * (onesArray[array[s]] - 1) *

```

```

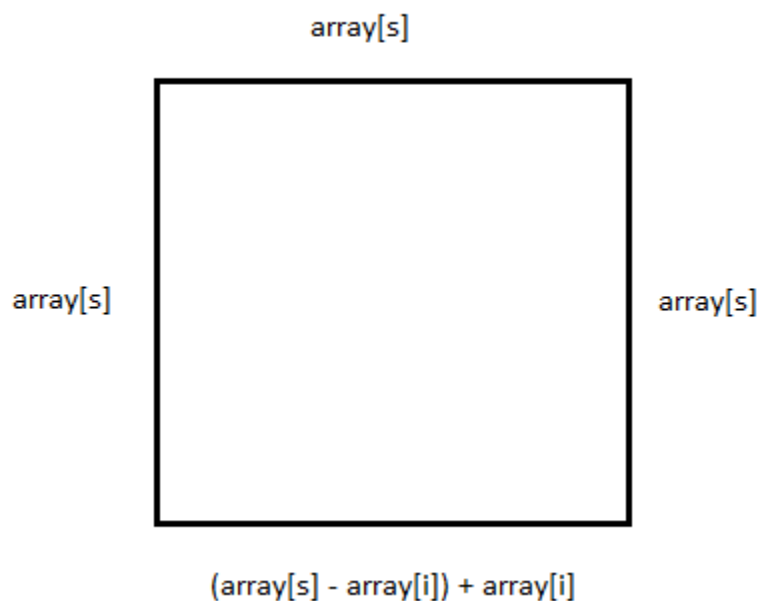
(onesArray[array[s]] - 2) / 6 *
pairsArray[array[s] - array[i]];
}

for (int j=1; j<i; j++) {
pairsArray[array[i] + array[j]]++;
}
}

```

Tablica pairsArray[] zawiera liczbę wystąpień kombinacji par o długościach danych indeksami.

W ten sposób mnożymy permutacje 3-elementowe wszystkich patyków o długości array[s], pary patyków sumujące się do długości (array[s]-array[i]). Po przeiterowaniu przez wszystkie pętle otrzymujemy ilość wszystkich kombinacji, danych rysunkiem:



Kluczowe okazuje się w tym rozwiązaniu odpowiednia implementacja algorytmu tak, aby pokrywał poprawnie przypadki z analizy pokrycia przypadków.

Kolejnym etapem jest znalezienie kombinacji, gdzie występują dwa patyki o tej samej długości, które pojedynczo tworzą bok. Należy rozważyć kilka różnych możliwości:

Fragment kodu:

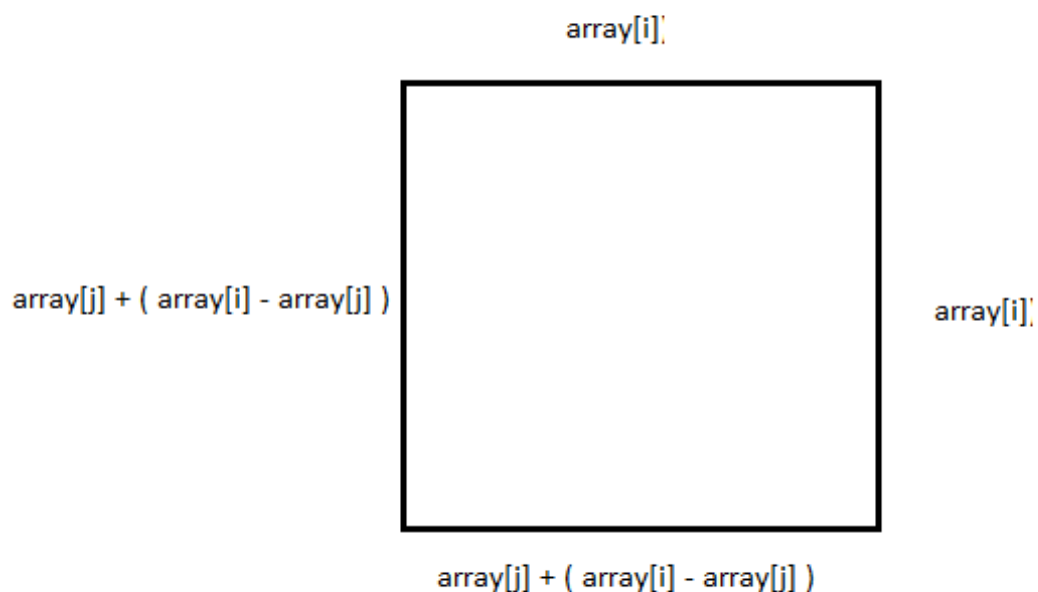
```
for (int i=1;i<=n;i++)
if (array[i]!=array[i-1] &&onesArray[array[i]]>1)
{
    DD=onesArray[array[i]]*(onesArray[array[i]]-1)/2;
D=0;
int j = 1;
for (j=1; array[i] >array[j] * 2; j++)
if (array[j]>array[j-1]){
    ans+=DD*onesArray[array[j]]*(onesArray[array[j]]-
1)/2*onesArray[array[i]-array[j]]*(onesArray[array[i]-array[j]]-1)/2;
ans+=DD*D*onesArray[array[j]]*onesArray[array[i]-array[j]];

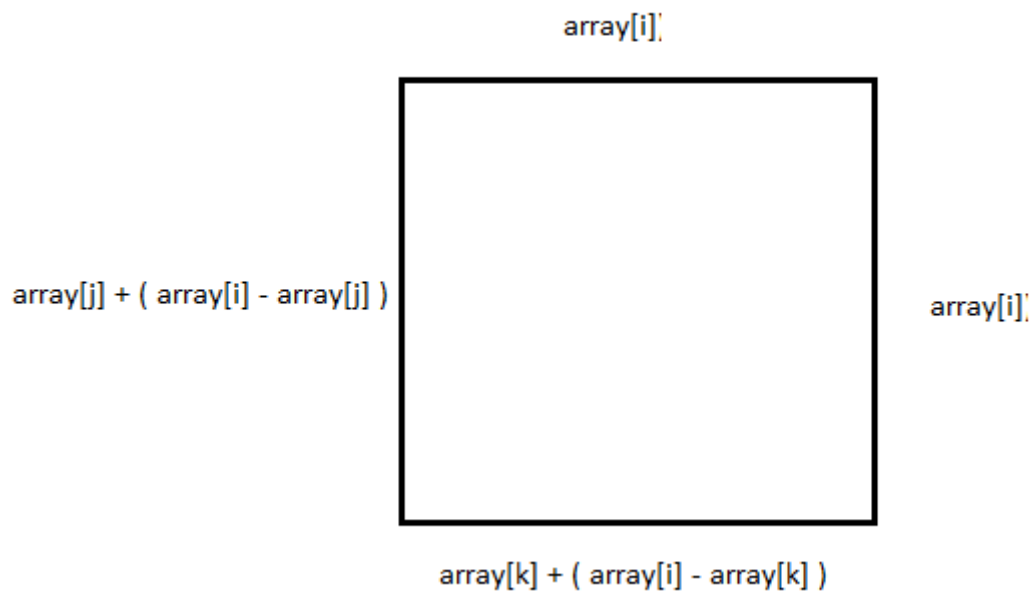
D+= onesArray[array[j]]*onesArray[array[i]-array[j]];

}
if (array[i]%2==0){
    ans+=DD*onesArray[array[i]/2]*(onesArray[array[i]/2]-
1)*(onesArray[array[i]/2]-2)*(onesArray[array[i]/2]-3)/24;
ans+=DD*D*onesArray[array[i]/2]*(onesArray[array[i]/2]-1)/2;
}
}
```


Zauważmy, że pętla zagnieżdżona jest wykonana jedynie dla elementów $\text{array}[j] < \text{array}[i]/2$, aby nie nastąpiły powtórzenia.

Analogicznie dla przypadku 'tripletów', należy tu liczyć dwumiany newtona w celu znalezienia unikalnych permutacji poszczególnych patyków o tych samych długościach. W pierwszym 'ifie' rozwiązane są przypadki dla :





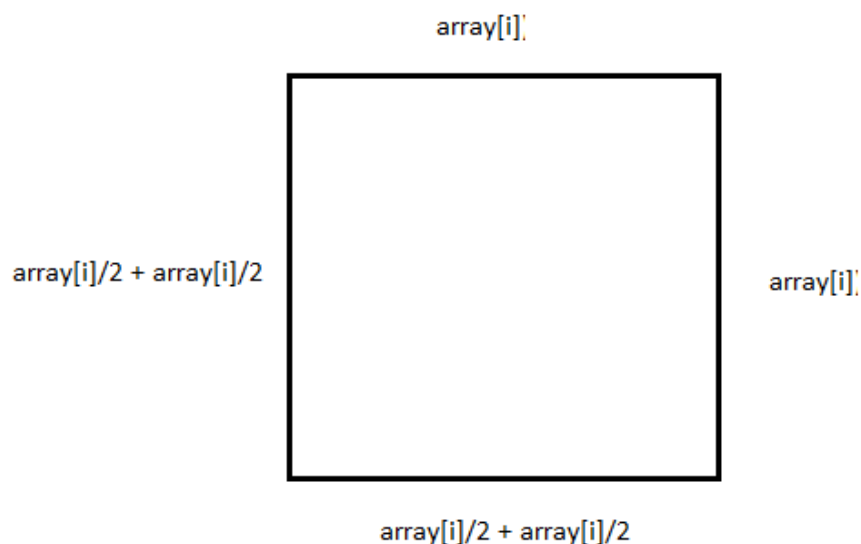
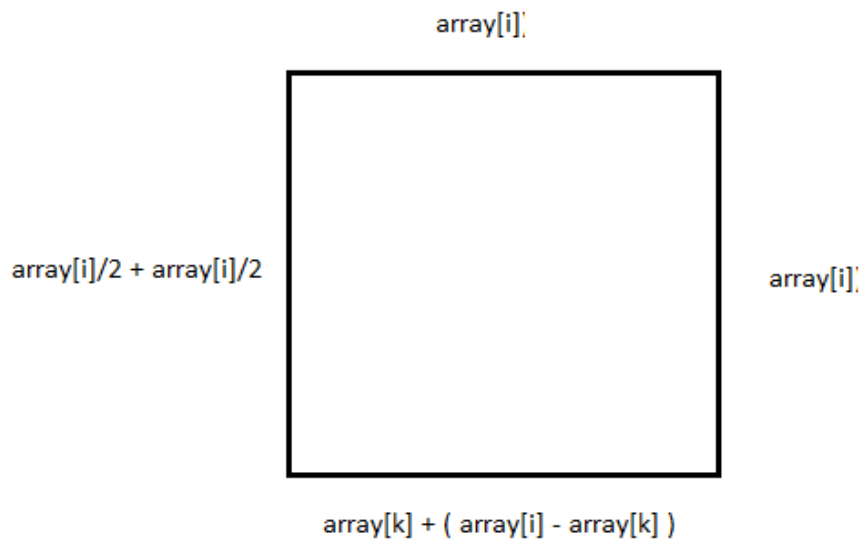
```
ans+=DD*onesArray[array[j]]*(onesArray[array[j]]-1)/2*onesArray[array[i]-array[j]]*(onesArray[array[i]-array[j]]-1)/2;
```

Pierwszy przypadek to poszukiwanie kombinacji, które składają się z dwóch patyków które pojedynczo tworzą bok (należy obliczyć dwumian newtona aby znaleźć unikalne permutacje), kombinacje dwóch patyków o długościach `array[j]` (ponownie dwumian newtona) oraz kombinacje dwóch patyków o długościach `array[i] - array[j]` (ponownie wykorzystać dwumian newtona).

```
ans+=DD*D*onesArray[array[j]]*onesArray[array[i]-array[j]];
```

Drugi przypadek jest bardzo podobny do poprzedniego, tyle że szukamy kombinacji par patyków `array[j]` oraz `array[i]-array[j]`, oraz wyliczonych we wcześniejszych iteracjach (czyli dla mniejszych `j`) par patyków `array[i]` oraz `array[i]-array[j]` (na rysunku dla odróżnienia przedstawionych jako `array[k] + (array[i]-array[k])`).

W drugim 'ifie; rozwiązane są przypadki:



```
ans+=DD*onesArray[array[i]/2]*(onesArray[array[i]/2]-1)*(onesArray[array[i]/2]-2)*(onesArray[array[i]/2]-3)/24;
```

Sprawdzamy dwie opcje. Pierwsza jest taka, że na utworzenie dwóch boków wykorzystamy cztery patyki tej samej długości (każdy $\text{array}[i]/2$). Dlatego musimy policzyć dwumian newtona w poszukiwaniu wszystkich unikalnych permutacji czterech patyków o tej samej długości.

```
ans+=DD*D*onesArray[array[i]/2]*(onesArray[array[i]/2]-1)/2;
```

Druga opcja to poszukiwanie dwu-elementowych unikalnych permutacji patyków dwa razy mniejszych od tych które pojedynczo tworzą bok, oraz kombinacje par patyków o długościach `array[j]` oraz `array[i] – array[j]`.

SŁÓW KILKA O WYPISANIU KOMBINACJI:

Kolejnym problemem było wyznaczenie jakich patyków użyć do zbudowania kwadratu. Dwa różne rozwiązania zastosowałem dla algorytmu brute force oraz dla algorytmu optymalnego.

Bruteforce:

W algorytmie brute force wykorzystuję wektor wskaźników do klas `Combinations`. Owe klasy przechowują długości/indeksy boków. Przy znalezieniu prawidłowej kombinacji za pomocą `push_back` (złożoność tej operacji to $O(1)$) wstawiam do wektora nową kombinację patyków. W tym podejściu przechowuję każdą kombinację patyków (również te które posiadają takie same boki, lecz są to tak naprawdę inne patyki). Chcąc uniknąć kosztu realokacji rozmiaru wektora, na początku programu zwiększam jego pojemność do dużej wartości (10^8) . Analizy złożoności algorytmu potwierdziły złożoność $O(n^6)$. Dla stosunkowo małych rozmiarów danych wejściowych (100 patyków) znalezienie rozwiązania trwa aż 25 sekund.

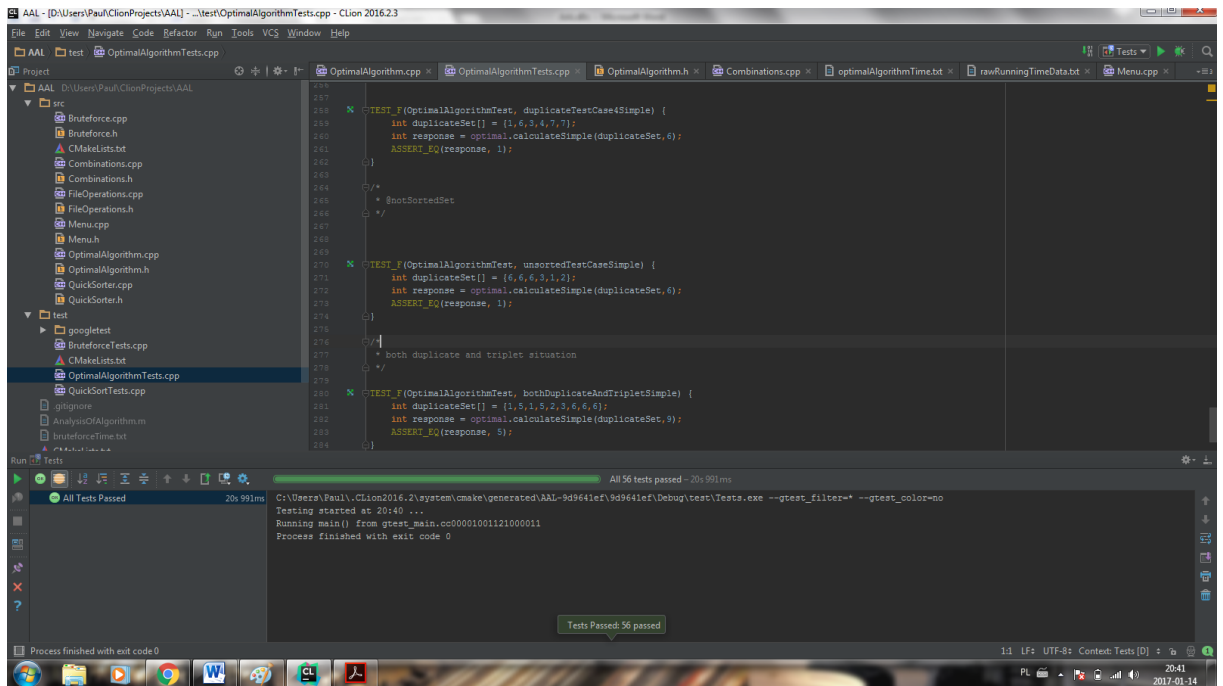
OptimalAlgorithm:

W algorytmie `OptimalAlgorithm`, podjąłem decyzję o nieużywaniu żadnej struktury danych, która stricte przechowywałaby wszystkie kombinacje patyków. Podjąłem próby (metoda `calculateSolutions()` klasy `OptimalAlgorithm`) przechowywania kombinacji w wektorze wskaźników do klas `Combinations`, jednakże przy testach dla dużych danych spowodowało to, błędy programu `std::bad_alloc`, których przyczyną było zabraknięcie miejsca na stercie z powodu zbyt dużej ilości zajmowanej przez wektor kombinacji patyków zbiorów (zazwyczaj błędy pojawiały się przy zbiorach danych większych niż około 3000 patyków) . Po drugie zdecydowałem się, że wypisywać będę wszystkie kombinacje, lecz różniące się przynajmniej jedną długością patyka (tzn. nie wypisuję kombinacji patyków o tych samych

długościach, które jednakże różnią się tym, że użyto różnych patyków). Wyświetlenie wyników następuje po wywołaniu metody `showCombinations` klasy `OptimalAlgorithm`. Działa ona bardzo podobnie do metody `calculateSimple` wyznaczającej liczbę rozwiązań. Ocena teoretyczna złożoności owej metody jest ciężka do przeprowadzenia, ze względu na silne skorelowanie danych wejściowych z wydajnością algorytmu. W metodzie występują identyczne pętle jak w metodzie `calculateSimple`, jednakże występuje również dodatkowa pętla zagnieżdżona, dla niektórych warunków. Ilość iteracji wspomnianej pętli nie jest bezpośrednio związana z rozmiarem danych wejściowych. Jest ona zmienna i zależna od występujących danych. Według mnie analiza teoretyczna złożoności wykazuje, że metoda powinna mieć złożoność nielepszą niż $O(n^2)$ i niegorszą niż $O(n^3)$. Metody `showCombinations` nie podaje analizie czasowej, ponieważ użycie wypisania wyników do `stdout` (`cout`), wypaczyłoby wyniki pomiarów czasu wykonania algorytmu.

TESTOWANIE:

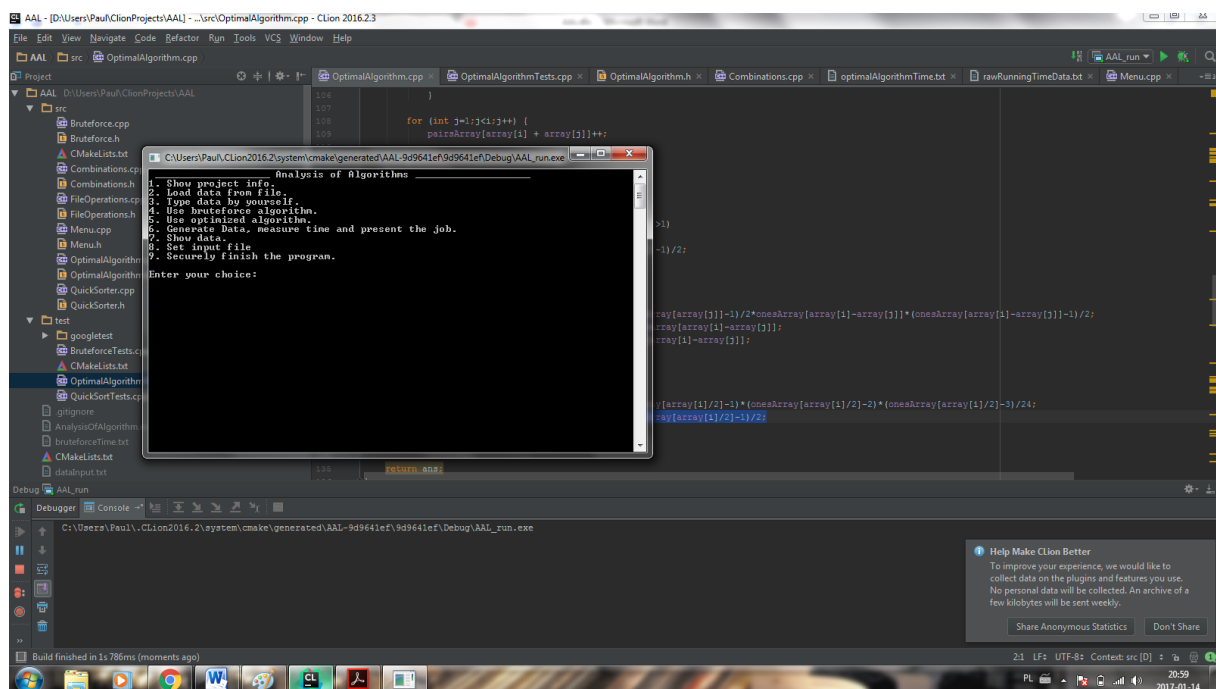
Kolejnym problemem jest testowanie poprawności działania algorytmu. Projekt prowadziłem w formule Test Driven Development. W osobnej ścieżce (folder `test`), znajdują się testy poprawności algorytmu. Osobno zostały przetestowane algorytmy sortowania, algorytm `bruteforce` oraz algorytm optymalny. Testowanie poprawności w wymienionych testach, opierało się na stosunkowo małych zbiorach danych. Mając pewność co do poprawności algorytmu `bruteforce`, przeprowadziłem również testy dla większych zbiorów danych dla algorytmu optymalnego, porównując czy odpowiedź algorytmu optymalnego dawała takie same wyniki jak `bruteforce`.



GUI:

Do współpracy z użytkownikiem przewidziałem prosty konsolowy interfejs graficzny z kilkoma opcjami. Zakładałam, że opcje same w sobie są proste i nie trzeba wyjaśniać ich działania:

GUI



ANALIZA ALGORYTMU TEORETYCZNA:

Dokonamy tutaj analizy złożoności zaproponowanego algorytmu.

Analizując kod programu (metoda **calculateSimple()** klasy **OptimalAlgorithm**),
spodziewamy się złożoności programu $O(n^2)$.

Dokonajmy dekompozycji metody **calculateSimple()**:

```
for (inti=1;i<=n;i++) array[i] = tab[i-1], onesArray[array[i]]++;
```

- Wypełnienie tablic, jedna pętla, złożoność **$O(n)$**

```
- quickSorter.sort(array,1,n,arrayOfIndices);
```

- Posortowanie zbioru, złożoność **$O(n \log n)$**

```
for (inti=1;i<=n;i++){  
for (ints=i+1;s<=n;s++)  
if (array[s]>array[s-1]) {  
ans += 111 * onesArray[array[s]] * (onesArray[array[s]] - 1) *  
(onesArray[array[s]] - 2) / 6 *  
pairsArray[array[s] - array[i]];  
}  
  
for (intj=1;j<i;j++) {  
pairsArray[array[i] + array[j]]++;  
}  
}
```

- Wyznaczanie kombinacji 'tripletów', złożoność **$O(n^2)$** .

```
- for (inti=1;i<=n;i++)  
if (array[i]!=array[i-1] &&onesArray[array[i]]>1)  
{  
DD=onesArray[array[i]]*(onesArray[array[i]]-1)/2;  
D=0;  
intj = 1;  
for (j=1; array[i] >array[j] * 2; j++)  
if (array[j]>array[j-1]){  
ans+=DD*onesArray[array[j]]*(onesArray[array[j]]-  
1)/2*onesArray[array[i]-array[j]]*(onesArray[array[i]-array[j]]-1)/2;  
ans+=DD*D*onesArray[array[j]]*onesArray[array[i]-array[j]];  
}}
```

```

D+= onesArray[array[j]]*onesArray[array[i]-array[j]];

}
if (array[i]%2==0) {
    ans+=DD*onesArray[array[i]/2]*(onesArray[array[i]/2]-
1)*(onesArray[array[i]/2]-2)*(onesArray[array[i]/2]-3)/24;
ans+=DD*D*onesArray[array[i]/2]*(onesArray[array[i]/2]-1)/2;
}
}

```

- Wyznaczanie kombinacji dla dwóch patyków o tych samych długościach, które pojedynczo tworzą bok. Złożoność $O(n^2)$.

WNIOSEK:

Summa summarum złożoność algorytmu, korzystając z Reguły sum nieiterowanych:

$$O(f(n), g(n), \dots) = O(\max(f(n), g(n), \dots)) = O(n^2).$$

ANALIZA ZŁOŻONOŚCI ALGORYTMU (RZECZYWISTE POMIARY)

Kolejnym etapem projektu jest przeprowadzenie analizy złożoności algorytmu. Pomiary czasu wykonuje korzystając z biblioteki **chrono**. Dla przeprowadzenia analizy przewidziałem następujące sekwencje w programie:

- Z Menu programu wybieramy opcję nr 6 – „Generate data, measure time and present job”
- W pliku rawRunningTimeData.txt pojawią się dane, które wykorzystuje skrypt napisany w języku Matlab
- Uruchamiamy skrypt AnalysisOfAlgoritm.m
- Wprowadzamy naszą predykcję co do złożoności algorytmu. ($O(n^x)$, gdzie x to nasza predykcja)

Przykładowe wyniki analizy:

Testy zostały wykonane dla rozmiarów zbioru
 $n \in \{5000, 10000, 15000, 20000, \dots, 95000\}$

Dla predykcji rozwiązania $O(n^2)$. Tabela 1

ALGORYTM Z ASYMPTOTĄ $O(T(n))$		
n	t(n) [ms]	q(n)
0	0	1.1035
5000	93	1.0293
10000	347	0.9690
15000	735	0.9566
20000	1290	0.9473
25000	1996	0.9815
30000	2978	0.9328
35000	3852	0.9409
40000	5075	1.0000
45000	6827	1.0000
50000	8428	0.9623
55000	9813	0.9659
60000	11722	0.9624
65000	13709	0.9906
70000	16363	0.9724
75000	18440	0.9537
80000	20577	0.9201
85000	22412	1.0052
90000	27450	1.1036
95000	33576	1.0231

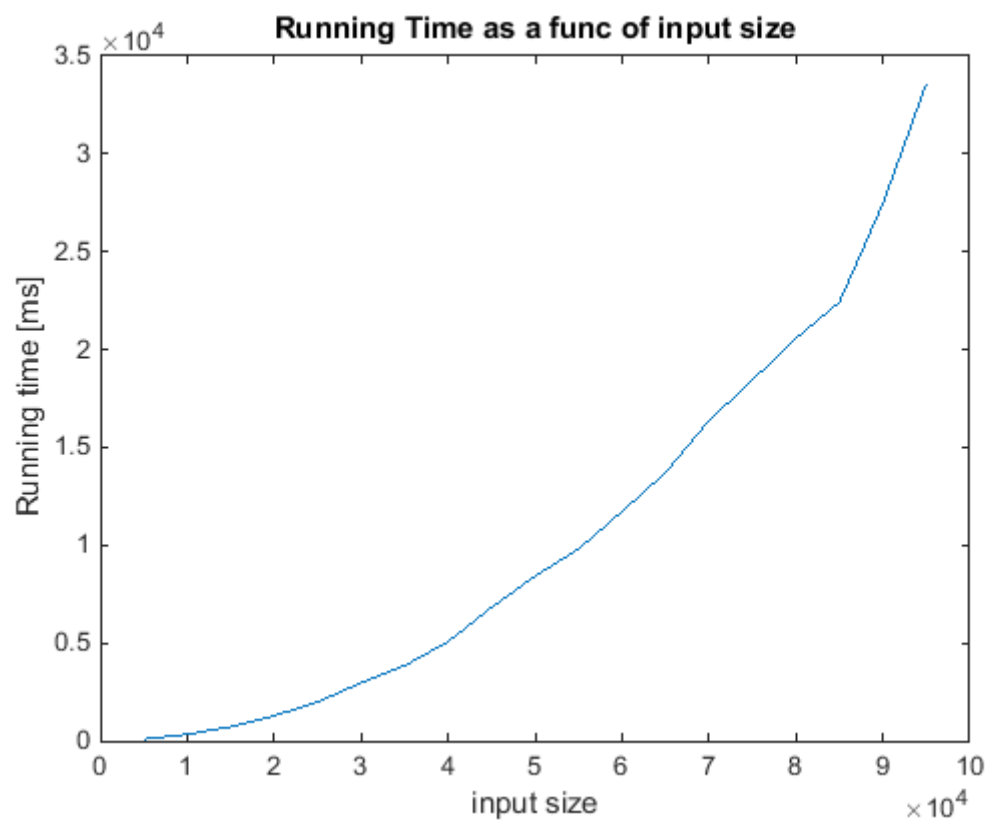
Z pliku optimalAlgorithmTime.txt

```
410  
411  
412  
413  
414 The Running time was: 0 ms for the input size of 0  
415 The Running time was: 93 ms for the input size of 5000  
416 The Running time was: 347 ms for the input size of 10000  
417 The Running time was: 735 ms for the input size of 15000  
418 The Running time was: 1290 ms for the input size of 20000  
419 The Running time was: 1996 ms for the input size of 25000  
420 The Running time was: 2978 ms for the input size of 30000  
421 The Running time was: 3852 ms for the input size of 35000  
422 The Running time was: 5075 ms for the input size of 40000  
423 The Running time was: 6827 ms for the input size of 45000  
424 The Running time was: 8428 ms for the input size of 50000  
425 The Running time was: 9813 ms for the input size of 55000  
426 The Running time was: 11722 ms for the input size of 60000  
427 The Running time was: 13709 ms for the input size of 65000  
428 The Running time was: 16363 ms for the input size of 70000  
429 The Running time was: 18440 ms for the input size of 75000  
430 The Running time was: 20577 ms for the input size of 80000  
431 The Running time was: 22410 ms for the input size of 85000  
432 The Running time was: 27450 ms for the input size of 90000  
433 The Running time was: 33576 ms for the input size of 95000
```

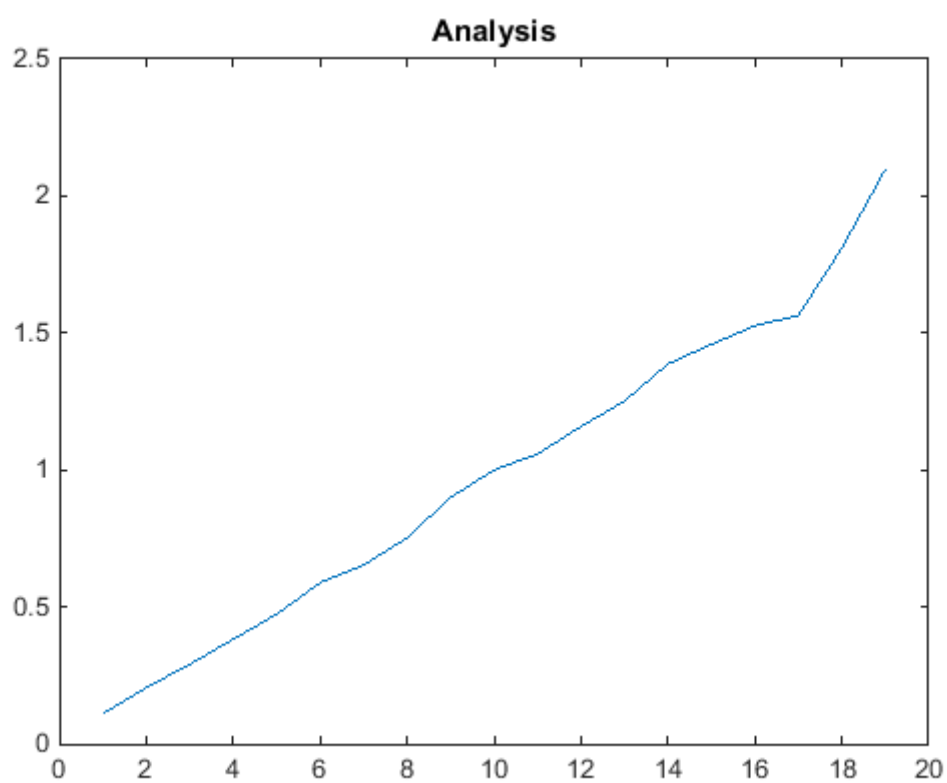
Run: AAL_run
Analysis of Algorithms
1. Show project info.
2. Load data from file.
3. Type data by yourself.
4. Use bruteforce algorithm.
5. Use optimized algorithm.
6. Generate Data, measure time and present the job.
7. Show data.
8. Set input file
9. Securely finish the program.
Enter your choice:

Build finished in 968ms (7 minutes ago)
4131 CRLF UTF-8 Context: <no context>
21:14
2017-01-14

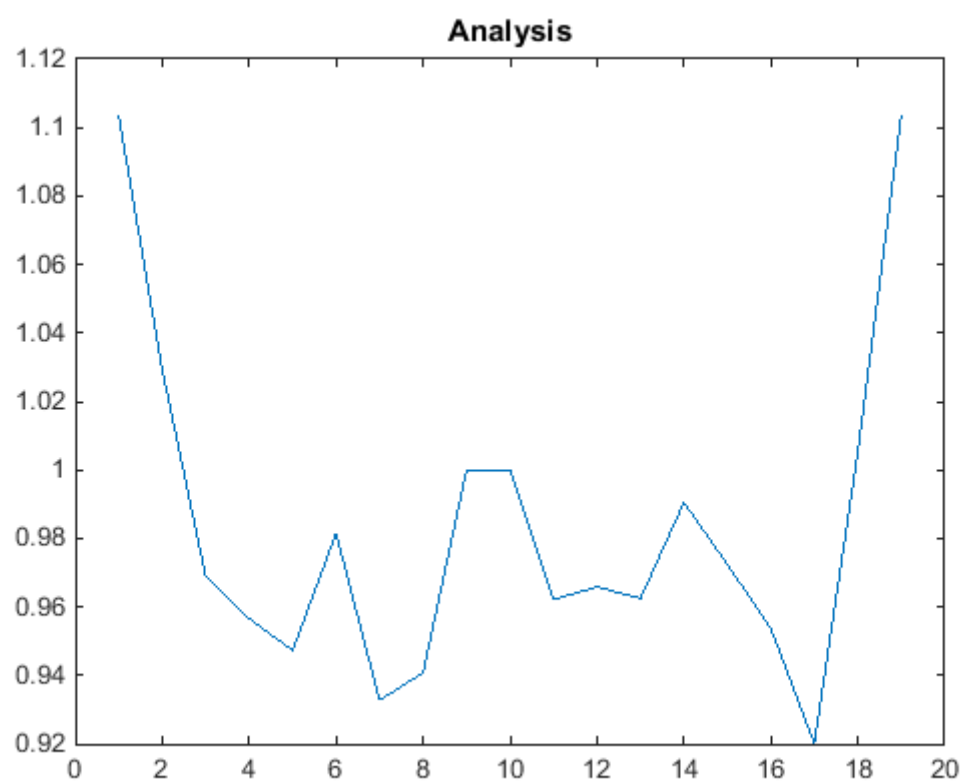
Help Make Clion Better
To improve your experience, we would like to collect data on the plugins and features you use. No personal data will be collected. An archive of a few kilobytes will be sent weekly.
Share Anonymous Statistics Don't Share



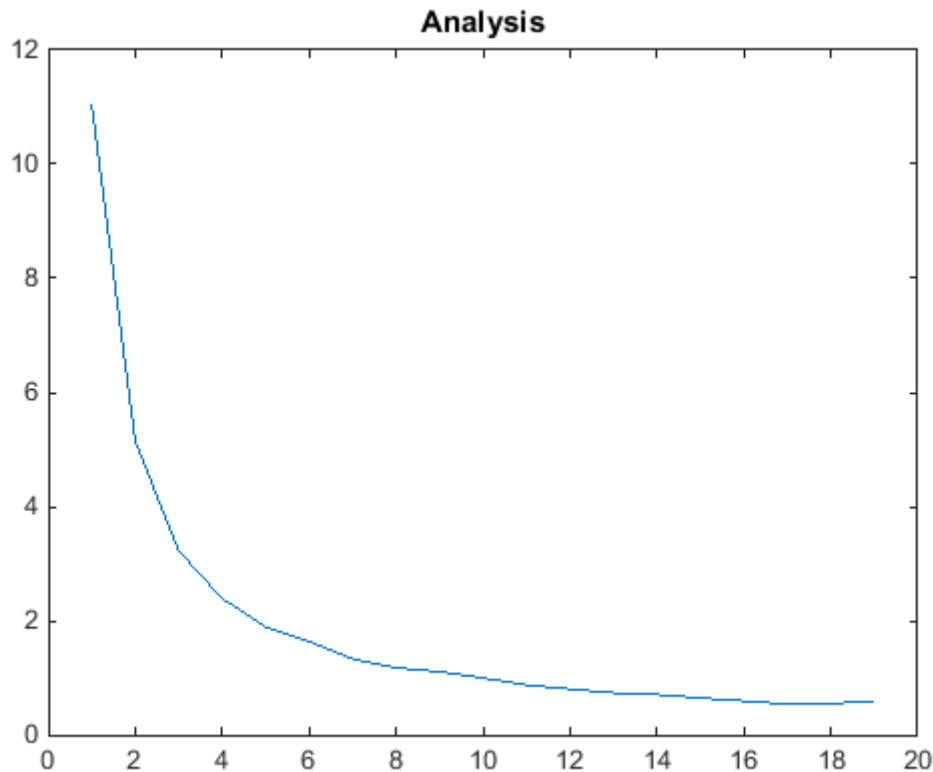
Predykcja dla $x = 1$ ($O(n^1)$).



Predykcja dla $x = 2$ ($O(n^2)$).



Predykcja dla $x = 3$ ($O(n^3)$).



WNIOSKI:

Analiza wykresu 'Running time as a func of input size' :

Kształt wykresu zdecydowanie najbardziej przypomina funkcję kwadratową.

Analiza wykresów 'Analysis':

Są to wykresy przedstawiające wyniki

$$q(n) = \frac{t(n)}{cT(n)} = \frac{t(n)T(n_{\text{mediana}})}{T(n)t(n_{\text{mediana}})}$$

Najlepsza predykcja to taka, gdzie współczynniki są bliskie 1. Przeszacowanie występuje gdy $q(n)$ malejące, zaś niedoszacowanie gdy $q(n)$ rosnące.

Analizując wykresy możemy bezpośrednio stwierdzić, że najlepsza jest predykcja rozwiązania o postaci $O(n^2)$, co potwierdza oceny teoretyczne.

LISTING KLAS I PLIKÓW WRAZ Z OBJAŚNIENIAMI:

- **src**

- **Bruteforce.h/Bruteforce.cpp**- zawiera klasę i implementację metod, której głównym zadaniem jest obliczanie ilości rozwiązań oraz przechowywanie możliwych kombinacji patyków (które z nich użyć) w wektorze klas Combinations.
- **Combinations.h/Combinations.cpp** – prosta klasa, która przechowuje długości (bądź indeksy) patyków dla wszystkich kombinacji.
- **FileOperations.h/FileOperations.cpp** – zawiera klasę, której zadaniem jest wykonywanie operacji na plikach (odczyt, zapis, modyfikacja).
- **Menu.h/Menu.cpp** – zawiera klasę i implementację metod, której zadaniem jest wyświetlanie Menu oraz interpretację żądań użytkownika.
- **OptimalAlgorithm.h/OptimalAlgorithm.cpp** – zawiera klasę i implementację metod, której zadaniem jest obliczanie ilości rozwiązań oraz wypisanie możliwych kombinacji patyków.
- **QuickSorter.h/QuickSorter.cpp** – zawiera klasę i implementację metod, której zadaniem jest posortowanie zbioru patyków metodą QuickSort.

- **test**

- **BruteforceTests.cpp** – testy poprawności algorytmu Bruteforce
- **OptimalAlgorithmTests.cpp** – testy poprawności algorytmu optymalnego
- **QuicksorterTests.cpp** – testy poprawności algorytmu sortowania

- **pliki**

- **dataInput.txt** – domyślnie plik zawierający zbiór patyków które chcemy wczytać do programu (Uwaga: pierwsza linijka pliku to rozmiar danych, następne to długości patyków).
- **optimalAlgorithmTime.txt** - czasy wykonania zadania dla poszczególnych rozmiarów zbiorów dla algorytmu optymalnego
- **bruteforceTime.txt** - czasy wykonania zadania dla poszczególnych rozmiarów zbiorów dla algorytmu bruteforce

- **rawRunningTimeData.txt** – dane o czasach wykonania w postaci do analizy dla Matlaba
- **folder główny**
 - **main.cpp** – uruchamia program
 - **AnalysisOfAlgorithm.m** – skrypt napisany w Matlab, do przeprowadzenia analizy złożoności.

GITHUB

Historia tworzenia projektu dostępna na moim profilu na **github**:

<https://github.com/Walczakp007/EITI-Projects/tree/master/AAL>