Politechnika Świętokrzyska w Kielcach

Wydział Elektroniki Automatyki i Informatyki

Mykola Bohdan

Eugeniusz Fedoroszczak

Sortowanie metodą Shella i Quicksort

2016

Sortowanie szybkie (quicksort)

1.1 Wprowadzenie

Podstawowa wersja algorytmu Quicksort została wynaleziona w 1960 roku przez C.A.R.Hoare'a. Algorytm jest popularny, bo nietrudno go zaimplementowac, działa sprawnie na danych róznego typu i czesto zuzywa mniej zasobów niz jakikolwiek inny algorytm.

Zalety:

• działa w miejscu, czyli "in situ" (uzywa tylko niewielkiego stosu pomocniczego)

• do posortowania n elementów wymaga srednio czasu proporcjonalnego do n * logn

• ma wyjatkowo skromna petle wewnetrza

Wady:

• jest niestabilny

• zabiera okolo n^2 operacji w najgorszym przypadku

• jest wrazliwy (prosty niezauwazony blad w implementacji moze powodowac niewlasciwe działanie w przypadku niektórych danych)

1.2 Zasada działania

Algorytm Quicksort jest metoda typu "dziel i rzadz". Bowiem jej dzialanie opiera sie na dzieleniu tablicy na dwie czesci, które nastepnie sortowane sa niezaleznie. Warto zaznaczyc, iz poczatkowy porzadek elementów w wejsciowym zbiorze danych ma wplyw na to, jak bedzie przebiegał podział. Nalezy wiedziec, ze to własnie proces podziału jest sednem metody. Możemy go opisac nastepujaco: dzielimy zbiór danych, wstawiajac pewien element rozgraniczajacy na jego własciwe miejsce, zmieniajac porzadek danych w tablicy tak, aby elementy mniejsze od niego znalazly sie z jego lewej strony, a wieksze - z prawej. Potem sortujemy rekurencyjnie lewa i prawa czesc tablicy. Poniewaz zawsze w wyniku procedury tworzacej podział przynajmniej jeden element trafia na własciwe miejsce, zatem takie rekurencyjne postepowanie w koncu doprowadzi do posortowania danych.

Ogólna strategia podziału:

1. wybieramy element rozgraniczajacy (wyznaczajacy podział), ten który trafi na własciwe miejsce

2. przegladamy tablice od jej lewego konca, az znajdziemy element wiekszy niz rozgraniczajacy

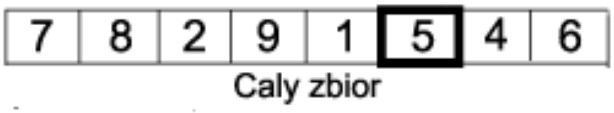
3. przegladamy tablice od jej prawego konca, az znajdziemy element mniejszy niz rozgraniczajacy

4. oba elementy, które zatrzymuja przegladanie tablicy, sa na pewno nie na swoich miejscach w tablicy, wiec je zamieniamy ze soba

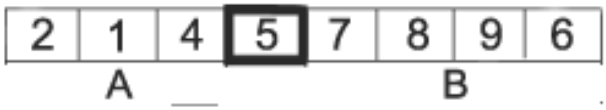
Postepujac w ten sposób upewniamy sie, ze zaden element po lewej stronie lewego wskaznika nie jest wiekszy od elementu rozgraniczajacego oraz zaden z prawej strony prawego wskaznika nie jest mniejszy niz element rozgraniczajacy, tak jak to widzimy na ponizszym rysunku:

Na powyzszym rysunku X oznacza wartosc elementu rozgraniczajacego, i jest lewym wskaznikiem, a j-prawym. Najlepiej jest zatrzymac przegladanie tablicy z lewej strony dla elementów wiekszych lub równych elementowi rozgraniczajacemu, a z prawej strony dla elementow mniejszych od niego lub mu równych. Kiedy wskazniki i oraz j mina sie, do zakonczenia całego procesu dzielenia tablicy trzeba juz tylko zamienic X ze skrajnycm lewym elementem prawego podzbioru (element wskazywany przez lewy wskaznik). Wewnetrzna petla sortowania szybkiego zwieksza wskaznik i porównuje element tablicy z ustalona wartoscia. To własnie ta prostota sprawia,ze quicksort jest tak szybki: trudno o prostrza petle wewnetrzna algorytmu sortowania.

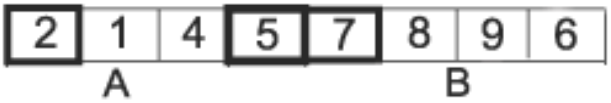
Zilustrujmy to prostym przykładem:



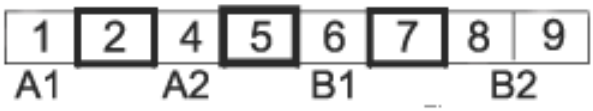
W zbiorze wybieramy jeden z elementów na element srodkowy



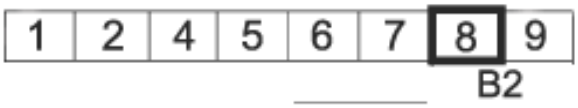
Pozostałe elementy umieszczamy odpowiednio po lewej stronie wybranego elementu, jesli sa od niego mniejsze lub równe lub po prawej stronie, jesli sa od niego wieksze. Otrzymujemy w ten sposób dwa przedzialy A i B



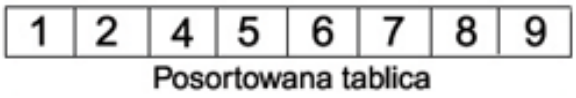
Te same operacje wykonujemy w kazdym przedziale. Odpowiada to rekurencyjnemu wywołaniu algorytmu dla kazdego z nich



Po przeniesieniu elementów na odpowiednie strony pozostanie tylko jednen przedzial dwuelementowy B2, który moze jeszcze byc nieposortowany. Pozostałe przedzialy A1, A2, B1 sa jednoelementowe i nie sortujemy ich dalej



Wybieramy element srodkowy - liczba 8



Podział daje przedzial jednoelementowy zawierajacy liczbe 9. Zatemalgorytm konczymy, zbiór został uporzadkowany.

1.3 Implementacje algorytmu Quicksort

Ponizej przedstawiono implementacje algorytmu sortowania:

```
PROCEDURE Quicksort(tablica, l, r)
  BEGIN
    IF l < r THEN      { jeśli fragment dłuższy niż 1 element }
      BEGIN
        i := PodzielTablice(tablica, l, r); { podziel i
zapamiętaj punkt podziału }
        Quicksort(tablica, l, i-1);     { posortuj lewą część }
        Quicksort(tablica, i+1, r);    { posortuj prawą część }
      END
 END{wybiera element, który ma być użyty do podziału
   i przenosi wszystkie elementy mniejsze na lewo od
   tego elementu, a elementy większe lub równe, na prawo
   od wybranego elementu }
  PROCEDURE PodzielTablice(tablica, l, r)
  BEGIN
     indeksPodzialu := WybierzPunktPodzialu(tablica, l, r);
{wybierz element, który posłuży do podziału tablicy}
     wartoscPodzialu := tablica[indeksPodzialu]; {zapamiętaj
wartość elementu}
     Zamien(tablica, indeksPodzialu, r); {przenieś element
podziału na koniec tablicy, aby sam nie brał udziału w
podziale}
     aktualnaPozycja := l;
     FOR i:=l; TO r-1 DO {iteruj przez wszytkie elementy, jeśli
element jest mniejszy niż wartość elementu podziału dodaj go do
"lewej" strony}
     BEGIN
         IF tablica[i] < wartoscPodzialu THEN
         BEGIN
             Zamien(tablica, i, aktualnaPozycja);
             aktualnaPozycja := aktualnaPozycja + 1;
         END
     END
     Zamien(tablica, aktualnaPozycja, r); {wstaw element
podziału we właściwe miejsce}
     return aktualnaPozycja;
  END { podstawowa implementacja wyboru punktu podziału -
wybiera element "środkowy" w tablicy }
  PROCEDURE WybierzPunktPodzialu(tablica, l, r)
  BEGIN
     return l + (r-l) div 2;
```

```
END{ zamienia miejscami elementy w komórkach i1, i2 }
PROCEDURE Zamien(tablica, i1, i2)
BEGIN
   aux := tablica[i1];
   tablica[i1] := tablica[i2];
   tablica[i2] := aux;
END
```

1.4 Złozonosc algorytmu Quicksort

Zarówno czas działania algorytmu sortowania szybkiego jak równiez zapotrzebowanie na pamiec sa uzaleznione od postaci tablicy wejsciowej. Od tego zalezy, czy podziały dokonywane w algorytmie sa zrównowazone, czy tez nie, a to z kolei zalezy od wybranego klucza podziału. W przypadku, gdy podziały sa zrównowazone, algorytm jest równie szybki jak np. sortowanie przez scalanie. Gdy natomiast podziały sa niezrównowazone, sortowanie moze przebiegac asymptotycznie tak wolno, jak sortowanie przez wstawianie.

Sortowanie metodą Shella

2.1 Wprowadzenie

Sortowanie Shella – algorytm sortowania działający w miejscu i korzystający z porównań elementów. Stanowi uogólnienie sortowania przez wstawianie, dopuszczające porównania i zamiany elementów położonych daleko od siebie. Jego pierwszą wersję opublikował w 1959 roku Donald Shell.

Złożoność czasowa sortowania Shella w dużej mierze zależy od użytego w nim ciągu odstępów. Wyznaczenie jej dla wielu stosowanych w praktyce wariantów tego algorytmu pozostaje problemem otwartym.

1.2 Zasada działania

Sortowanie Shella to algorytm wieloprzebiegowy. Kolejne przebiegi polegają na sortowaniu przez proste wstawianie elementów oddalonych o ustaloną liczbę miejsc *h*, czyli tak zwanym *h*-sortowaniu.

Poniżej zilustrowano sortowanie przykładowej tablicy metodą Shella z odstępami 5, 3, 1.

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
dane wejściowe:	62	83	18	53	07	17	95	86	47	69	25	28
po 5-sortowaniu:	17	28	18	47	07	25	83	86	53	69	62	95
po 3-sortowaniu:	17	07	18	47	28	25	69	62	53	83	86	95
po 1-sortowaniu:	07	17	18	25	28	47	53	62	69	83	86	95

Pierwszy przebieg, czyli 5-sortowanie, sortuje osobno przez wstawianie zawartość każdego z fragmentów (a_1, a_6, a_{11}), (a_2, a_7, a_{12}), (a_3, a_8), (a_4, a_9), (a_5, a_{10}). Na przykład fragment (a_1, a_6, a_{11}) zmienia z (62, 17, 25) na (17, 25, 62).

Następny przebieg, czyli 3-sortowanie, sortuje przez wstawianie zawartość fragmentów (a_1, a_4, a_7, a_{10}), (a_2, a_5, a_8, a_{11}), (a_3, a_6, a_9, a_{12}).

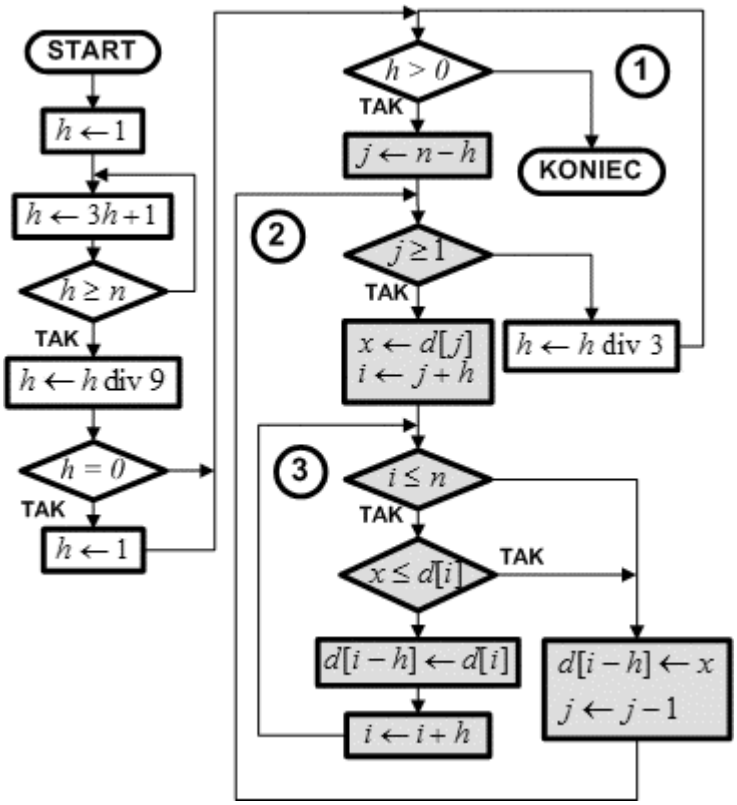
Ostatni przebieg, czyli 1-sortowanie, to zwykłe sortowanie przez wstawianie całej tablicy (a_1,..., a_{12}).

Jak widać, fragmenty tablicy, na których operuje algorytm Shella, są z początku krótkie, a pod koniec dłuższe, ale prawie uporządkowane. W obu tych przypadkach sortowanie przez proste wstawianie działa wydajnie.

Sortowanie Shella nie jest stabilne, czyli może nie zachowywać wejściowej kolejności elementów o równych kluczach. Wykazuje ono zachowanie naturalne, czyli krótszy czas sortowania dla częściowo uporządkowanych danych wejściowych.

Schemat blokowy

Algorytm sortowania metodą Shella jest ulepszonym algorytmem sortowania przez wstawianie. Aby się o tym przekonać, wystarczy spojrzeć na schemat blokowy. Kolorem szarym zaznaczyliśmy na nim bloki, które dokładnie odpowiadają algorytmowi sortowania przez wstawianie. Jedyną modyfikacją jest wprowadzenie odstępu h zamiast liczby 1.



Na początku algorytmu wyznaczamy wartość początkowego odstępu h. Wykorzystujemy tu sugestie prof. Donalda Knutha.

Po wyznaczeniu h rozpoczynamy pętlę warunkową nr 1. Pętla ta jest wykonywana dotąd, aż odstęp h przyjmie wartość 0. Wtedy kończymy algorytm, zbiór będzie posortowany.

Wewnątrz pętli nr 1 umieszczony jest opisany wcześniej algorytm sortowania przez wstawianie, który dokonuje sortowania elementów poszczególnych podzbiorów wyznaczonych przez odstęp h. Po zakończeniu sortowania podzbiorów odstęp h jest zmniejszany i następuje powrót na początek pętli warunkowej nr 1.

1.3 Implementacje algorytmu Shella

Ponizej przedstawiono implementacje algorytmu sortowania:

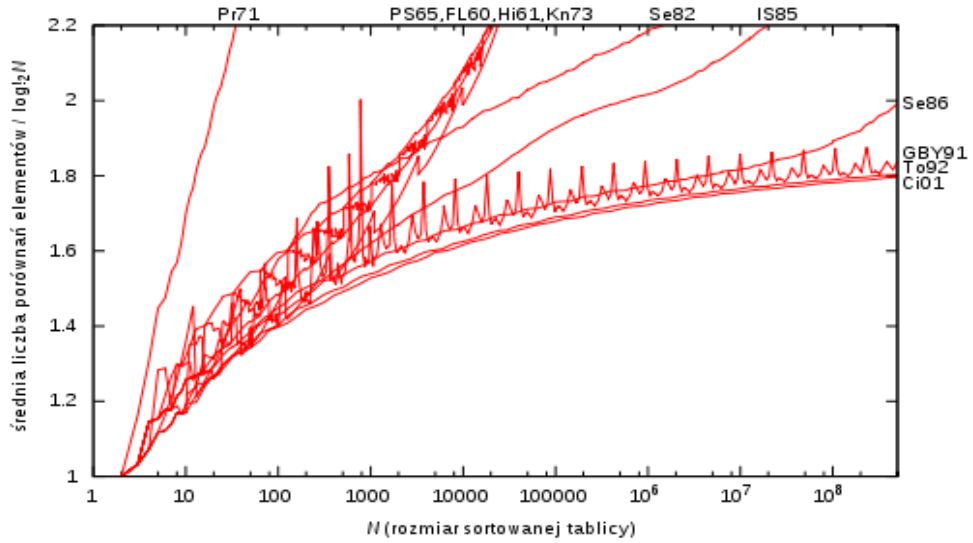
```
void shellsort (int v[], int n)
{
    int gap, i, j, temp;
    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j= i -gap; j>=0 && v[j]>v[j +gap]; j-= gap) {
                temp = v[j];
                v[j] = v[j + gap];
```

```
                v[j + gap] = temp;
            }
}
```

1.4 Złozonosc algorytmu Shella

Na podstawie doświadczeń odgadnięto, że z ciągami Hibbarda i Knutha algorytm działa w średnim czasie rzędu $O(N^{5/4})$, a z ciągiem Gonneta i Baezy-Yatesa wykonuje średnio $0{,}41N\ln N(\ln\ln N+1/6)$ przesunięć elementów. Aproksymacje średniej liczby operacji czynione kiedyś dla innych ciągów zawodzą, gdy sortowane tablice liczą miliony elementów.

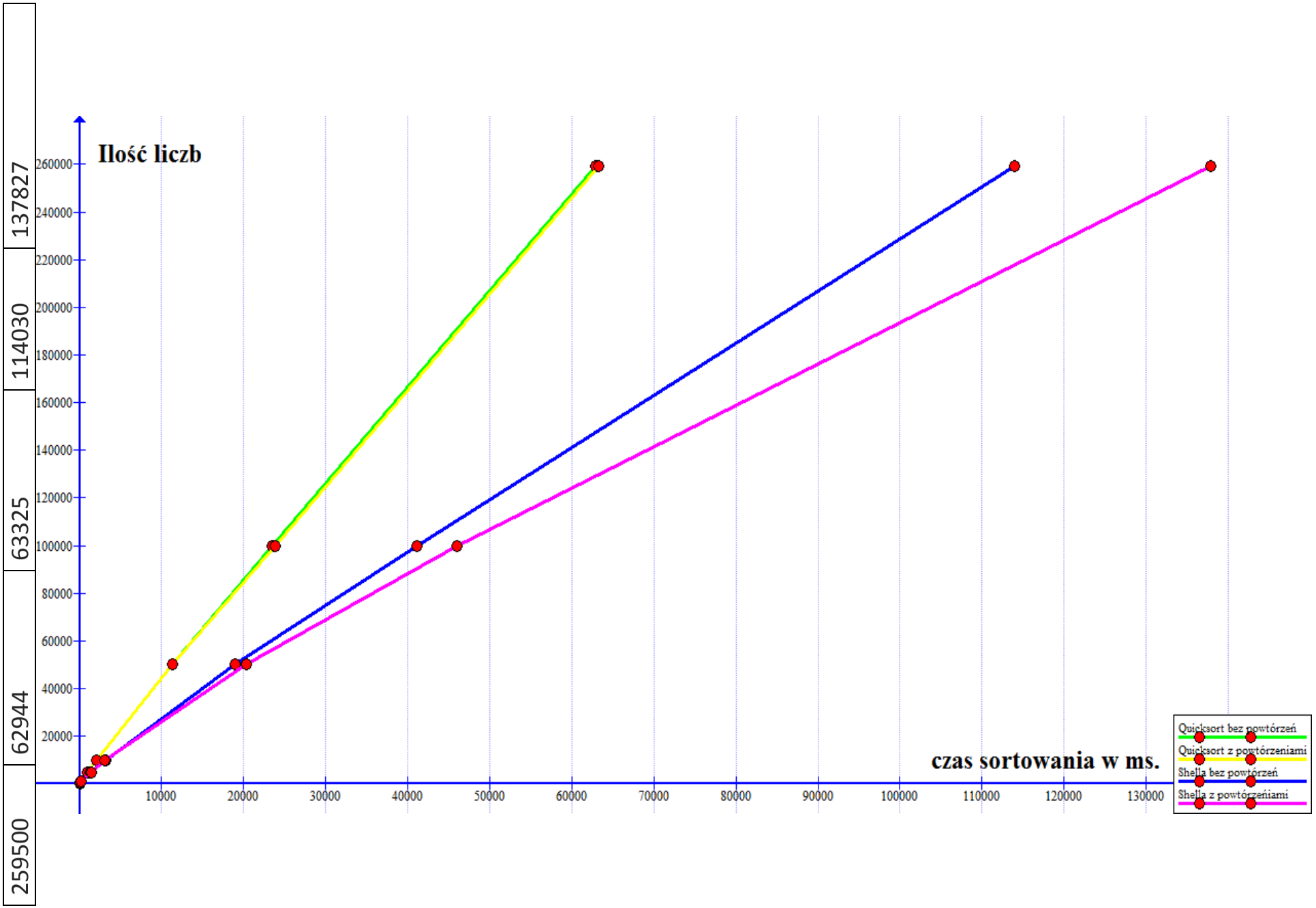
Poniższy wykres przedstawia średnią liczbę porównań elementów w różnych wariantach sortowania Shella, dzieloną przez teoretyczne minimum, czyli $\log_2 N!$, przy czym do ciągu 1, 4, 10, 23, 57, 132, 301, 701 dodano dalsze wyrazy zgodnie ze wzorem $h_k = \lfloor 2{,}25h_{k-1} \rfloor$.



3. Porównanie na bazie naszego kodu.

Zakladamy że 2266 takty procesora to 1 ms.

Ilość elemenyów	Quicksort bez powtórzeń(ms.)	Quicksort z Powtarzeniami	Shella bez powtórzeń	Shella z Powtórzeniami (ms.)
10	1	1	0	0
50	6	6	5	5
100	13	14	11	12
500	83	80	92	89
1000	172	170	200	201
5000	973	960	1358	1373
10000	2020	2056	3144	3065
50000	11270	11281	18939	20386
100000	23401	23803	41137	45938


259500
62944
63325
114030
137827
Ilość liczb
260000
240000
220000
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
110000
120000
130000
czas sortowania w ms.
Quicksort bez powtórzeń
Quicksort z powtórzeniami
Shella bez powtórzeń
Shella z powtórzeniami

Na grafice można zaobserwować, że algorytm Quicksort jest szybszy podczas sortowania 500 liczb, przy czym, jeżeli liczby powtarzają się, to czas sortowania wydłuża się w stosunku do liczb nie posiadających powtorzeń. Algorytm Shella zaś jest szybszy na małych porcjach danych, które w przybliżeniu wynoszą 500 .

4.Kod programy.

Programa sklada się z 3 części to: generator liczb:

```
void generator()
{
    char c;
    long MAS[n],array[n+1],k=0,i=0,g=0,u=n;
    int rdm=32767,j=1;
    for (g=0; g<n+1; g++)
    {
        array[g]=0;
    }
    printf("\nLiczby musza powtarzac sie? y/n: ");
    scanf("%c",&c);
    while(c!='y' && c!='n' && c!='Y' && c!='N')
    {
        scanf("%c",&c);
    }
    if(c=='n' || c=='N')
    {
        k=rand()%n;
        for (i=0; i < n; i++)
        {
            while(1)
            {
                if(u<rdm*j)
                {
                    k=rand()%(n-rdm*(j-1))+1+rdm*(j-1);
                    if(u<rdm*j)j--;
                }
                else j++;
                if ( array[k] == 0 )
                {
                    array[k] = 1;
                    MAS[i] = k;
                    u--;
                    break;
                }
```

```
            }
        }
    }
    else
    {
        for (i=0; i < n; i++)
        {
            MAS[i] = rand()%RAND_MAX;
        }
    }
}
```

Zgenerowany ciąg zapisuje do pliku gen_ciag.txt

Algorytm sortowania Shella:

```
int shell(long *MAS)
{
    int h,i,j,x;
    for(h = 1; h < n; h = 3 * h + 1);
    h /= 9;
    if(!h)
        h++;
    while(h)
    {
        for(j = n - h - 1; j >= 0; j--)
        {
            x = MAS[j];
            i = j + h;
            while((i < n) && (x > MAS[i]))
            {
                MAS[i - h] = MAS[i];
                i += h;
            }
            MAS[i - h] = x;
        }
        h /= 3;
    }
    return *MAS;
}
```

Wyniki sortowania zapisuje do pliku sort_Shell_ciag.txt

Algorytm sortowania Quicksort:

```
int quick(long *MAS,long lewy,long prawy)
{
    long v=MAS[(lewy+prawy)/2];
    long i, j, x;
    i=lewy;
    j=prawy;
    do
    {
        while(MAS[i]<v) i++;
        while(MAS[j]>v) j--;
        if (i<=j)
        {
            x=MAS[i];
            MAS[i]=MAS[j];
            MAS[j]=x;
            i++;
            j--;
        }
    }
    while (i<=j);

    if(j>lewy) quick(MAS,lewy,j);
    if(j<prawy) quick(MAS,i,prawy);
    return *MAS;
}
```

Posortowani liczby zapisuje po pliku quick_sort_ciag.txt

6.Wnioski

Z wyników działania naszego programu doszliśmy do następujących wniosków.

Quick Sort jest, jak sama nazwa wskazuje, bardzo szybkim algorytmem sortującym. Jest to jeden z najpopularniejszych algorytmów sortowania.Jego złożoność czasowa wynosi O(n log n). Sortowanie Quicksort opiera się na technice "dziel i zwyciężaj". Jest to najszybszy algorytm sortujący z tej klasy. Dodatkowo jest to algorytm sortujący w miejscu, czyli nie zajmuje dużo pamięci. Jeżeli uruchomimy program na komputerze w dużą mocą obliczeniową efektywniej jest użyć sortowania Quicksort, ponieważ wywołuje się rekurencyjnie. Mając komputer o niższym taktowaniu CPU i nie zależy nam na czasie operacji - lepiej użyć sortowania Shella.