Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie		
Wstęp do Informatyki		
Wydział: **EAIiIB**	*Kierunek:* **Informatyka**	*Imię i nazwisko:* **Krzysztof Stasiowski**
Rok i semestr: **2017/2018, I**	*Data ćwiczenia:* **2017-12-06**	

1 Cel Ćwiczenia

Celem Ćwiczenia jest napisanie programu zawierającego funkcje implementujące 6 metod sortowania (bąbelkowe, przez wstawianie, przez wybieranie, szybkie, przez scalanie, przez kopcowanie). Każda z funkcji powinna być wywołana raz dla 50 elementowej tablicy liczb całkowitych. Wyjściem z programu powinno być 7 linii - tablica nieposortowana, oraz posortowana każdą z 6 metod.

Następnie należy przeprowadzić eksperyment,porównujący zaimplementowane algorytmy sortowania.

Wyniki przedstaw w formie tabeli zbiorczej oraz wykresów przedstawiających zależność czasu wykonania od rozmiaru danych dla każdej z metod sortowania.

2 Implementacja Algorytmów

2.1 Algorytmy o złożoności obliczeniowej $O(n^2)$

Implementacja Algorytmów z tej kategorii nie sprawiła mi większych problemów. Funkcje sortowania składają się co najwyżej z kilku linijek kodu. Podczas implementacji sortowania Bąbelkowego(kod.2) zaimplementowałem optymalizację związaną z dodaniem flagi isSorted, co pozwala na zmniejszanie czasu działanie programu, jeżeli sortowana tablica jest częściowo uporządkowana.

2.2 Algorytmy o złożoności obliczeniowej $O(nlog(n))$

2.2.1 Quicksort

Implementacja Tego algorytmu zajęła mi więcej czasu niż implementacja poprzednich. Podstawowa funkcja quick sort była prosta w implementacji. Jednak implementacja funkcja pomocnicza partitionQS(kod.4) zajęła mi więcej niż powinna z powodu trudnych do zdiagnozowania błędów. Ostatecznie napisałem tę funkcję od początku bez większych problemów i źródło oryginalnych błędów pozostaje dla mnie tajemnicą.

2.2.2 Mergesort

Implementacja Tego algorytmu była podobna trudnościowo do algorytmu quickSort. Podobnie jak w przypadku quicksort, to funkcja mrege była trudniejsza. (kod.5) Do trudności w implementacji można zaliczyć moją próbę zaimplementowania tego sortowania w miejscu, co powodowało utratę danych w niektórych przypadkach.

Podobnie niewielkie trudności sprawiło mi również wyliczenie ilości elementów w łączonych tablicach, i przekopiowanie ich wartości do tablicy tymczasowej. Z powodu tego że pogubiłem się w tym, czy powinienem brać pod uwagę ostatni element tablicy, czy nie.

2.2.3 Heapsort

Zdecydowanie największe trudności sprawiło mi zaimplementowanie Heapsort(kod.6). Połączony brak doświadczenia z operowaniem na kopcach i moje początkowe błędne podejście do tematu budowy kopca, spowodowało że moja implementacja wielokrotnie działała błędnie i nieprzewidywanie.

Początkowo chciałem zaimplementować budowę kopca od początku. Jednak po paru próbach okazało się, że prostszą i szybszą metodą jest budowa kopca od tyłu.

Funkcja Naprawy kopca również przeszła parę iteracji, pierwsza rekurencyjna implementacja, miała problem z dojściem do warunku końcowego,oraz również tutaj parokrotnie miałem problem z indeksami początku i końca.

W końcowej implementacji funkcja ta naprawia kopiec od root do momentu gdy dany node nie posiada dziecka, bądź jest w prawidłowej konfiguracji,ta implementacja wydaje mi się być najbardziej optymalna jaką mogę napisać.

2.2.4 Implementacja w Python

Listing 1: sortowanie przez wybieranie

```
def selectonSort(n):
    for x in range(len(n)):
        #wybieramy najmniejszą wartość(jej index) w zakresie (x-końca tabli
        min = x
        for i in range(x,len(n)):
            if(n[min]>n[i]):
                min=i

        #zamieniamy miejscami najmniejszy element z elementem x#
        n[min],n[x] = n[x],n[min]

    return n
```

Listing 2: sortowanie bąbelkowe

```
def bubbleSort(n):
    isSorted = False
    sortingCount = 1;
    size = len(n)
    while(not isSorted and sortingCount <=size):
        #Zmieniamy flage w nadzieji że posortowaliśmy tablice#
        isSorted = True
        #sprawdzamy elementy od 0 do SortCount#
        for i in range(size-sortingCount):
            if(n[i] > n[i+1]):#Jeżeli znaleźliśmy#
            #nieposortowane elementy zamieniamy je miejscami#
                n[i], n[i+1] = n[i+1],n[i]
                #Ustawiamy flagę#
                isSorted = False
        #zwiększamySortCount#
        sortingCount+=1
    return n;
```

Listing 3: sortowanie przez wstawianie

```
def insertionSort(n):
    for i in range(1,len(n)):
        #pobieramy kolejny element#
        insVal = n[i];
        #sprawdzamy czy jest mniejszy od już pobranych elementów#
        j = i-1;
        while (j>=0 and n[j]>insVal):
            #przesówamy większe  elementy w prawo#
            n[j+1]=n[j]
            #przechodzimy do elementu j-1#
            j-=1
        #w tym momencie jesteśmy nad elementem mniejszym od wstawianej w
        #mamy większe wartości przesunięte w prawo#
        # i bezpośrednio na prawo od nas jest skopiowany element od nas
        n[j+1]=insVal

    return n
```

Listing 4: sortowanie szybkie

```
def partitionQS(tab,start,end):

    #wybieramy pivot#
    pivot = random.randint(start,end-1);
    tab[pivot],tab[start] = tab[start],tab[pivot]
    #przemieszczamy pivot na początek #
    pivot =start;#aktualizujemy pozycję #

    i=start #ustawiamy licznik liczbmniejczych od pivot,#
    # będzie to miejsce w którym ostatecznie wyląduje pivot#
    for j in range(start+1,end):#przechodzimy prez liczby poza pivot#
        if(tab[j]<=tab[pivot]):
            i+=1#zwiększamy licznik liczb większych od pivot#
            tab[i],tab[j] = tab[j],tab[i]#namieniamy liczbę mniejszą od
            #z liczbą, która znajduje sie pod licznikiem#

    tab[pivot],tab[i] = tab[i],tab[pivot]
    #ustawiamy pivot na swoim miejscu#
    return i #zwracamy miejsce na którym ustawiliśmy pivot#

def quickSort(tab,start,end):
    if(start <end):
        pivot =partitionQS(tab,start,end)
        quickSort(tab,start,pivot)
        quickSort(tab,pivot+1,end)

    return tab
```

Listing 5: sortowanie przez łączenie

```
    i = start #ustawiamy index na początek łązonyvh tablic#

    Ai = 0 #index tablicy a#
    Al = (pivot-start)+1#długość tablicy a#
    Bi = 0 #index tablicy b#
    Bl = (end-pivot) #długość tablicy b#

    #wstawiamy najmniejsze elemnty tablicy A i B#
    #aż skończą się elemnty jednej z tablic#
    while(Ai<Al and Bi<Bl):
            #sprawdzamy któryz elentów jest mniejszy#
            if(tTab[Ai] < tTab[Al+Bi]):
                tab[i]=tTab[Ai]#dodajemy element z tablicy A#
                i+=1#przesówamy index łączonej tablicy#
                Ai+=1#przesówamy index tablicy A#
            else:
                tab[i]=tTab[Al+Bi] #dodaemy element tablicy B#
                i+=1
                Bi+=1
    #dodajemy nie dodane elementy tablicy A#
    while(Ai<Al):
        tab[i]=tTab[Ai]
        i+=1
        Ai+=1
    #dodajemy niedodane elementy tablicy B#
    while(Bi<Bl):
        tab[i]=tTab[Al+Bi]
        i+=1
        Bi+=1

def mergeSort(tab,start,end):
    if(end>start):
        pivot = math.floor((start+end)/2)
        mergeSort(tab,start,pivot)
        mergeSort(tab,pivot+1,end)
        merge(tab,start,pivot,end)

    return tab
```

Listing 6: sortowanie przez Kopcowanie

```
    return math.floor((n-1)/2)
def getLeft(n):
    return 2*n+1
def getRight(n):
    return 2*n+2
# funkcja pomocnicza zamiany elementów#
def swap(tab,i,j):
    tab[i],tab[j] = tab[j],tab[i]

def repairHeap(tab,start,end):
    root = start; #pierwszym rodzicem jest pobracy rodzic#

    while(getLeft(root) <= end):
    #wykonujemy do momentu aż rozic nie ma co najmniej 1 dziecka#

        swaper =root #ustawiamy index elmentu do zmiany na rodzica#
        if(tab[getLeft(root)] > tab[root]):
        #sprawdzamy czy rodzic jest większy od 1 dziecka#
            swaper=getLeft(root)

        if(getRight(root) <=end and tab[swaper] < tab[getRight(root)]):
            #sprawdzamy czy rodzic ma 2 dziecko#
            #i czy jest większe od elemty pod swaper#
            #(1 dzieko , rodzic)#
            swaper =getRight(root)

        #jeżeli rodzic jest mniejszy od któregoś z dzieci#
        if( not root == swaper):
            swap(tab,swaper,root) #zamieniamy je miejscami#
            root = swaper #i przechodzimy do naprawy kopca zamienionego
        else:
            break # inaczej kończymy wykonywanie#

def buildHeap(tab):
    size = len(tab)
    parent = getParent(size-1) #pobieramy ostatniego rodzica kopca
#

    while(parent >= 0):
        repairHeap(tab,parent,size-1) #naprawiamy warunek kopca dla niego
        parent-=1 #pobieramy poprzedniego rodzica kopca#

def heapSort(tab):
    buildHeap(tab) # budujemy kopiec#
    i = len(tab)-1 #pobieramy ostatni element kopca , #
    # jest on jednocześnie ostatnim elementem tablicy#

    while(i>0):
        swap(tab,0,i) #zamieniamy ostatni elemenent tablicy z pierwszym#
        #pierwszy element jest największy#
        i-=1 #ostani element tablicy jest posortowany ,#
        #sortujemy pozostałe elementy#
        repairHeap(tab,0,i) #naprawiamy kopiec dla pozostałych elementów#
        #powyższe wykonujemy aż przejdziemy przez wszyskie elementy tabli

    return tab

def selectonSort(n):
```

3 Porównanie Algorytmów

3.1 Algorytmy $O(n^2)$

Algorytmy z tej grupy wyraźnie odstają od algorytmów o korzystniejszej złożoności. Dla małych wartości rozmiaru danych, są porównywalne jednak stosunkowo szybko ich czas działania rośnie.

3.1.1 Bubblesort

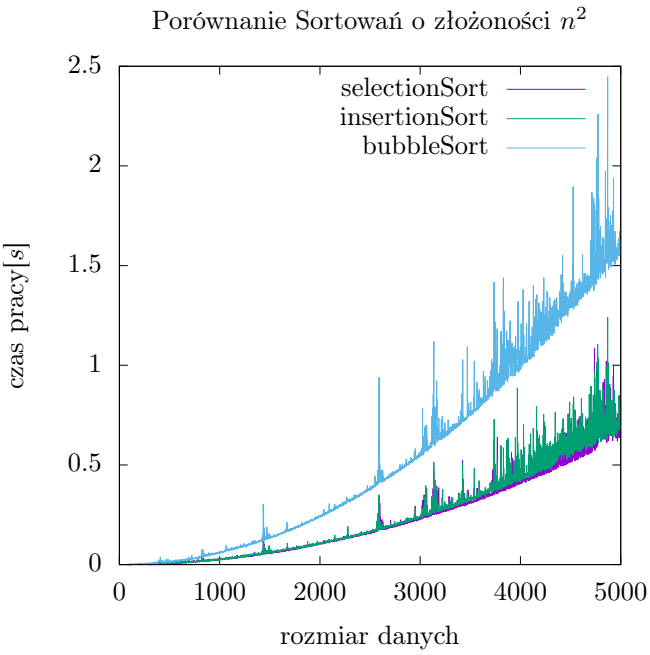
Najgorszy z porównywanych algorytmów, rośnie zdecydowanie szybciej niż Insertion i Selection sort.

3.1.2 Insertionsort

Rośnie w sposób prawie Identyczny Selection sort, jednak jest minimalnie wolniejszy.

3.1.3 Selectionsort

Najszybszy z pośród algorytmów o złożoności $O(n^2)$, jednak jeżeli chodzi o skalowanie, nie dotyka nawet najwolniejszego algorytmu z kategorii $O(nlog(n))$



3.2 Algorytmy $O(nlog(n))$

3.2.1 Heapsort

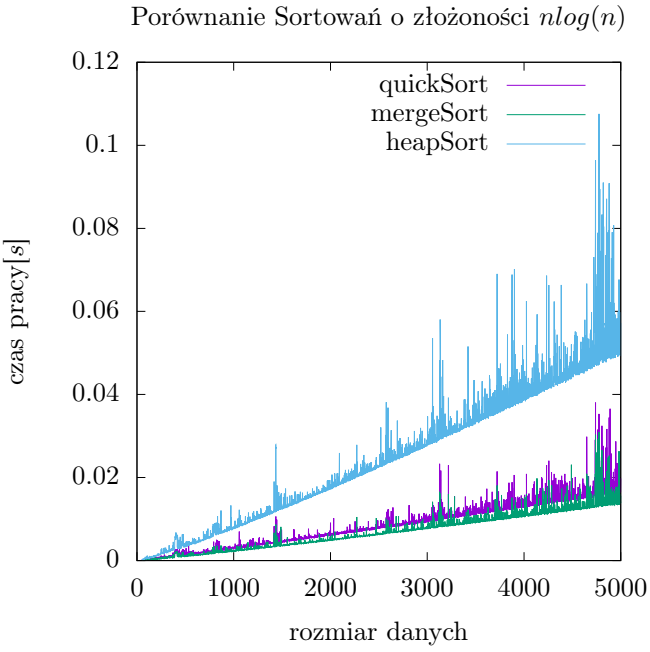
Najwolniejszy spośród Algorytmów $O(nlog(n))$, ale i tak jest zdecydowanie szybszy od wszystkich algorytmów z poprzedniej kategorii.

3.2.2 Quicksort

Prawie równie szybki co mergesort, jak wskazuje na to nazwa jest bardzo szybki.

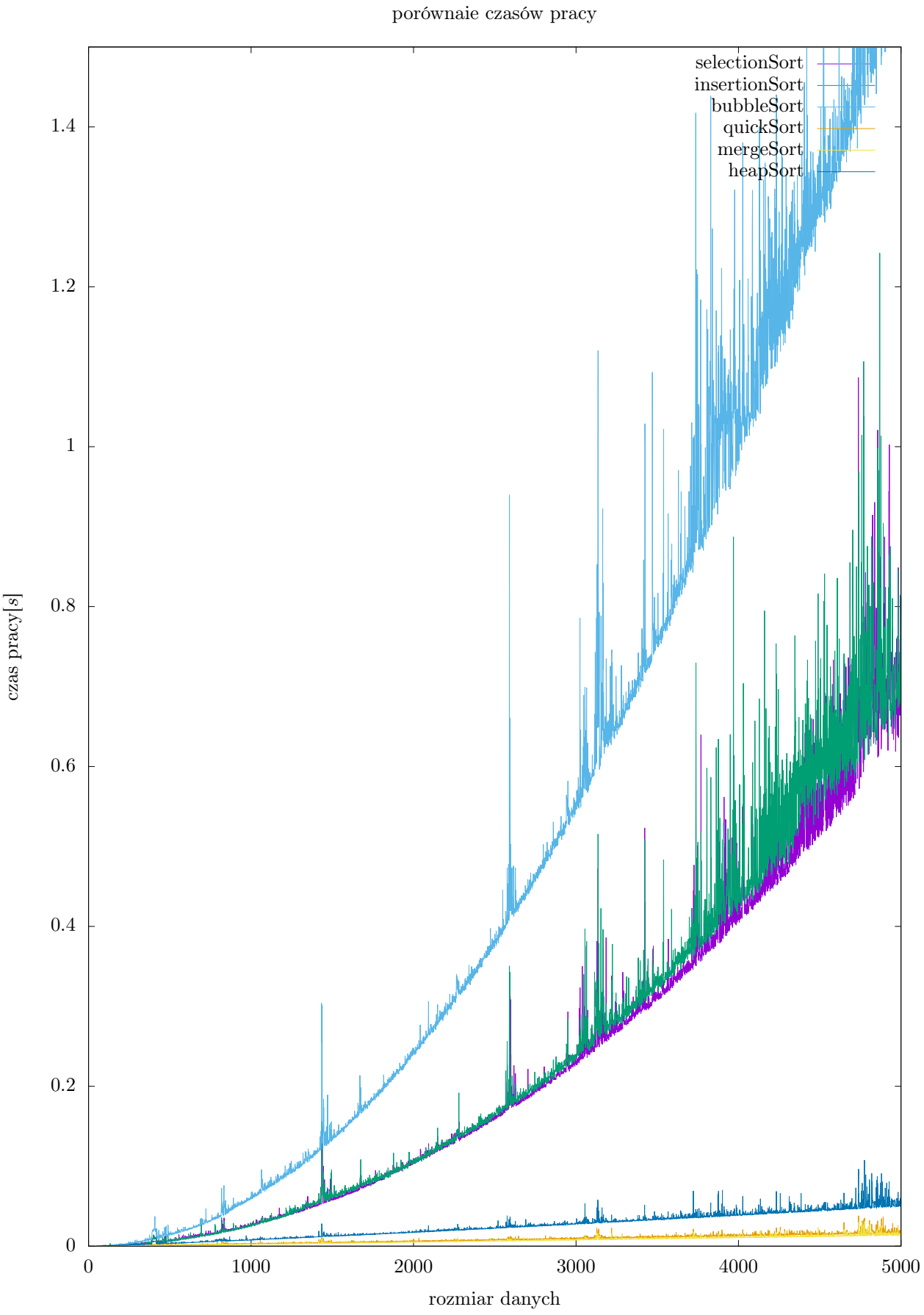
3.2.3 Mergesort

Najszybszy spośród algorytmów z tej grupy. Jest szybszy od quicksort, jednak niewiele.



3.3 Graf Zbiorowy

1



4 Tabela Danych

Ilość Danych	Selection Sort	Insertion Sort	Bubble Sort	Quick Sort	Merge Sort	HeapSort
000050	8.2969666e-05	6.7234039e-05	1.3756752e-04	1.2707710e-04	9.0360641e-05	2.1743774e-04
000213	1.0871887e-03	1.0857582e-03	2.2888184e-03	6.9999695e-04	5.0330162e-04	1.3210773e-03
000378	3.7508011e-03	3.8356781e-03	9.1915131e-03	1.3840199e-03	9.6845627e-04	3.1986237e-03
000543	9.1092587e-03	8.7084770e-03	1.7965078e-02	1.5451908e-03	1.1613369e-03	3.7691593e-03
000708	1.3277054e-02	1.4138222e-02	2.8710127e-02	2.0649433e-03	1.6288757e-03	5.2256584e-03
000873	1.9816637e-02	1.9558668e-02	4.5598745e-02	2.5749207e-03	1.9683838e-03	6.8838596e-03
001038	3.0313492e-02	2.9367924e-02	6.4420223e-02	3.1034946e-03	2.5217533e-03	8.1777573e-03
001203	3.8374662e-02	3.6315441e-02	8.8618994e-02	3.5316944e-03	2.8200150e-03	1.0079384e-02
001368	4.7637701e-02	5.0462723e-02	1.1014962e-01	4.2123795e-03	3.2434464e-03	1.1162281e-02
001533	6.0076952e-02	6.1381102e-02	1.4341331e-01	4.8534870e-03	3.7250519e-03	1.2858629e-02
001698	7.4801922e-02	7.7743053e-02	1.7754602e-01	5.1672459e-03	4.1561127e-03	1.4416695e-02
001863	9.0390444e-02	9.3888998e-02	2.0639682e-01	5.8312416e-03	4.6041012e-03	1.6146421e-02
002028	1.0534835e-01	1.0884619e-01	2.4791336e-01	6.2737465e-03	5.0113201e-03	1.7662525e-02
002193	1.2383056e-01	1.2909031e-01	2.9116940e-01	7.2736740e-03	5.5322647e-03	1.9274235e-02
002358	1.4106202e-01	1.4576268e-01	3.3726692e-01	7.4536800e-03	6.0098171e-03	2.1260738e-02
002523	1.6239643e-01	1.7416286e-01	3.7915826e-01	7.7915192e-03	6.4592361e-03	2.2706032e-02
002688	1.8450785e-01	1.9095469e-01	4.5141816e-01	8.2139969e-03	6.9358349e-03	2.4534702e-02
002853	2.0670390e-01	2.1484780e-01	4.9019861e-01	8.9535713e-03	7.4193478e-03	2.6071787e-02
003018	2.2889280e-01	2.4500179e-01	5.6051660e-01	9.5853806e-03	7.7993870e-03	2.8541327e-02
003183	2.5926518e-01	2.6745486e-01	6.4496946e-01	1.0535479e-02	9.4540119e-03	3.0462980e-02
003348	2.9580784e-01	2.9762959e-01	6.8329930e-01	1.0490894e-02	8.8644028e-03	3.1525850e-02
003513	3.1271553e-01	3.3725929e-01	7.6366305e-01	1.2103319e-02	9.3114376e-03	3.5044432e-02
003678	3.5328698e-01	3.6938119e-01	8.4064198e-01	1.1670589e-02	9.7613335e-03	3.5134315e-02
003843	3.8712263e-01	3.9141130e-01	9.2527032e-01	1.6978264e-02	1.0441780e-02	3.7187815e-02
004008	4.4321918e-01	4.4693470e-01	1.0033290e+00	1.3342381e-02	1.0665894e-02	3.8694620e-02
004173	4.6335316e-01	5.2783680e-01	1.1233208e+00	1.3784170e-02	1.1221409e-02	4.1190386e-02
004338	4.9272728e-01	5.4786468e-01	1.2658055e+00	1.5193701e-02	1.1820793e-02	4.2471409e-02
004503	5.5529881e-01	7.1020341e-01	1.3581100e+00	1.5171528e-02	1.2264252e-02	4.6386719e-02
004668	5.9948516e-01	6.3994598e-01	1.3855681e+00	2.0328760e-02	1.3112068e-02	4.6194077e-02
004833	6.3262343e-01	7.2919226e-01	1.4850950e+00	1.6600132e-02	1.3568401e-02	4.8115492e-02
004998	6.8900585e-01	7.9403973e-01	1.6175060e+00	1.6523838e-02	1.4181852e-02	4.9433708e-02
005164	7.0259261e-01	7.1649146e-01	1.6791706e+00	1.8723488e-02	1.4376163e-02	5.6964397e-02
005329	9.6525240e-01	9.1719627e-01	2.1944273e+00	2.1457672e-02	1.8300772e-02	6.8377018e-02
005494	8.6968303e-01	8.6476231e-01	2.0379345e+00	1.9347906e-02	1.7659903e-02	5.9159040e-02
005659	8.3820367e-01	8.8606143e-01	2.0765920e+00	2.1866322e-02	1.6323090e-02	8.6206198e-02
005824	9.7409129e-01	1.1750064e+00	2.3777277e+00	2.1214247e-02	1.6854525e-02	6.1924934e-02
005989	9.9388456e-01	1.0387962e+00	2.5561547e+00	2.5100946e-02	2.2998095e-02	8.4253788e-02
006154	1.0123808e+00	1.0481517e+00	2.6700680e+00	3.0823231e-02	2.1936417e-02	8.2845926e-02
006319	1.0198622e+00	1.1019258e+00	2.6237936e+00	2.1128893e-02	1.7672062e-02	6.5348387e-02
006484	1.2046494e+00	1.2476697e+00	4.3319948e+00	3.1078815e-02	2.1356106e-02	1.3769674e-01