Dokumentacja końcowa

przedmiot: Analiza Algorytmów
autor: Damian Mazurkiewicz

Treść zadania

Każdy egzemplarz pewnego urządzenia składa się z k różnych części. Są one zgromadzone, jako nieuporządkowana sekwencja, w długim magazynie. Uporządkować je tak, aby utworzyły k-elementowe, wewnętrznie uporządkowane komplety, potrzebne do zbudowania kolejnych egzemplarzy urządzenia. Jeśli liczby poszczególnych części są nierówne, części nadmiarowe mogą pozostać nieuporządkowane. Części są przenoszone za pomocą suwnicy, zawsze grupami po k sąsiadujących ze sobą części, z dowolnego miejsca sekwencji na jej koniec, bez zmiany ich uporządkowania, a następnie końcowa część sekwencji jest dosuwana do przodu, aby wypełnić powstałą lukę. Ułożyć plan pracy suwnicy, dyktujący w kolejnych ruchach, którą grupę k części przemieścić na koniec sekwencji. Wykonać jak najmniejszą liczbę ruchów. Porównać czasy obliczeń i wyniki różnych metod.

Oznaczenia:
k - ilość części składowych każdej maszyny.
n - ilość części na linii produkcyjnej.
r - ilość nieuszeregowanych elementów.

Struktury danych

Do rozwiązania problemu potrzebne mi było użycie struktury danych, która reprezentowałaby bieżący stan posortowania na linii produkcyjnej. Wybrałem do tego szablon **std::list** ze standardowej biblioteki C++. Wybór uzasadniam tym, że używając listy, operacja przeniesienia części na koniec linii produkcyjnej będzie miała złożoność stałą (gwarantuje to operacja **splice** mająca złożoność stałą gdy przenosimy elementy w obrębie jednej listy). Ewentualnymi wadami tego rozwiązania byłby brak dostępu do dowolnego elementu w czasie mniejszym niż liniowy, aczkolwiek używane algorytmy nie wymagają losowego dostępu do elementów tylko iterowania się po nich.
Pojedyncze elementy reprezentuję jako zmienne typu **int**, gdyż jest to najbardziej intuicyjne rozwiązanie, elementy są posortowane w obrębie kompletu, gdy stanowią ciąg arytmetyczny, którego pierwszym wyrazem jest 0, a różnica wynosi 1.

Rozwiązanie I

Założenie: Algorytm gwarantuje prawidłowe posortowanie, gdy liczba nadmiarowych elementów wynosi co najmniej **k**. Jeżeli ten warunek nie będzie spełniony, algorytm posortuje minimalnie **n-k** elementów. Poniżej algorytm przedstawiony jest w pseudokodzie:

```
n = line.size(); //Ilość elementów na linii produkcyjnej.
k = components.size(); //Ilość elementów przypadających na 1 pakiet.

currentPosition = line.start();
currentElement = 0;

//Iterujemy się kolejno po pozycjach na linii produkcyjnej.
for (; currentPosition.first < n - k;
increment(currentPosition),currentElement = (currentElement + 1) % k) {

        //W tym przypadku nie trzeba nic przestawiać, gdyż na tym miejscu
jest dobry element.
        if (*currentPosition.second == currentElement) {
            continue;
        }

        Position explorer = currentPosition;
        //Szukamy elementu do wstawienia na bierzącą pozycję.
            for (; explorer.second!=line.end(); increment(explorer))
                if (*explorer.second == currentElement)
                    break;
        }

      //Jeśli nie znaleziono potrzebnego elementu, dalsze sortowanie nie ma
sensu.
        if(explorer.second==line.end())
            break;

      //Ustawiamy element tak, by między nim, a pozycją na którą chcemy go
przenieść była liczba elementów podzielna przez k.
        explorer = correctPosition(explorer);

      //Przenosimy element na bierzącą pozycję, poprzez przestawianie tych
elementów, które są pomiędzy, na koniec.
        int movesToDo = (explorer.first - currentPosition.first) / k;
        for (; movesToDo > 0; --movesToDo) {
            moveToEnd(currentPosition);
        }

    }
```

Rozwiązanie II

To rozwiązanie jest usprawnionym rozwiązaniem pierwszym, usprawnienie polega na różnicy w szukaniu elementu, który chcemy przestawić na bieżącą pozycję, w tym algorytmie przeszukujemy listę produkcyjną do końca i wybieramy element, który ma za sobą najdłuższy ciąg posortowanych elementów (jednak nie większy niż k, gdyż takiego i tak nie dałoby się w całości przenieść). Pozwala to zmniejszyć liczbę ruchów suwnicy.

Założenie: Algorytm gwarantuje prawidłowe posortowanie, gdy liczba nadmiarowych elementów wynosi co najmniej **k**. Jeżeli ten warunek nie będzie spełniony, algorytm posortuje minimalnie **n-k** elementów.
Złożoność obliczeniowa: **$O(n^2)$**

```
n = line.size(); //Ilość elementów w linii produkcyjnej.
k = components.size(); //Ilość elementów przypadających na 1 pakiet.

currentPosition = line.start();
currentElement = 0;

//Iterujemy się kolejno po pozycjach na linii produkcyjnej.
for (; currentPosition.first < n - k;
increment(currentPosition),currentElement = (currentElement + 1) % k) {

        //W tym przypadku nie trzeba nic przestawiać, gdyż na tym miejscu
jest dobry element.
        if (*currentPosition.second == currentElement) {
            continue;
        }

        Position explorer = currentPosition;
        //Szukamy elementu do wstawienia na bierzącą pozycję. Poszukujemy
pozycji, za którą znajduje się możliwie najdłuższy ciąg posortowanych
elementów.
            Position bestChoice = line.end();
            int bestChoiceLength = 0;
            for(;explorer.second!=line.end(); increment(explorer))
            {
                if(*explorer.second==currentElement)
                {
                    int length = 1;
                    int elementMatch = (currentElement+1)%k;
                    Position measure = explorer;
                    increment(measure);

                  //Sprawdzamy ile posortowanych elementów jest za tą
                  pozycją.
                    for(;length<k && measure.second!=line.end();
increment(measure), elementMatch=(elementMatch+1)%k)
                    {
```

```
                        if(*measure.second==elementMatch)
                            length++;
                        else
                            break;
                    }
                  //Jeżeli ciąg posortowanych elementów jest większy niż
dotychczas najdłuższy, to przypisujemy go jako obecnie najdłuższy.
                    if(length>bestChoiceLength)
                    {
                        bestChoice=explorer;
                        bestChoiceLength=length;
                    }
                }
            }
            explorer=bestChoice;


      //Jeśli nie znaleziono potrzebnego elementu, dalsze sortowanie nie ma
sensu.
        if(explorer.second==line.end())
            break;

      //Ustawiamy element tak, by między nim, a pozycją na którą chcemy go
przenieść była liczba elementów podzielna przez k.
        explorer = correctPosition(explorer);

      //Przenosimy element na bierzącą pozycję, poprzez przestawianie tych
elementów, które są pomiędzy, na koniec.
        int movesToDo = (explorer.first - currentPosition.first) / k;
        for (; movesToDo > 0; --movesToDo) {
            moveToEnd(currentPosition);
        }

    }
```

Szacowanie złożoności asymptotycznej

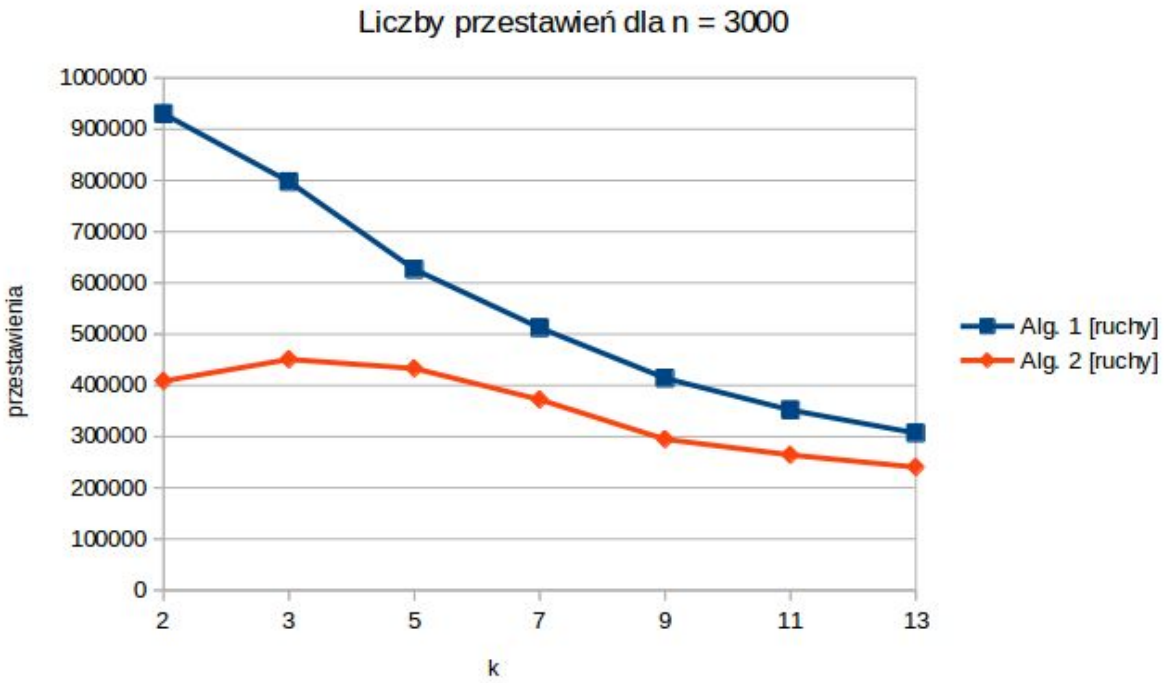
Oba algorytmy mają taką samą pesymistyczną złożoność i wynosi ona **$O(n^2)$**. W zewnętrznej pętli iterujemy się po niemal wszystkich **n** elementach na linii produkcyjnej. Następnie wewnątrz tej pętli wykonujemy trzy czynności:

1. Szukamy elementu do przestawienia
   Ilość elementów do sprawdzenia w poszukiwaniu elementu do przestawienia w algorytmie pierwszym nie zależy od n, ponieważ szukamy pierwszego elementu, który nam pasuje (czyli zależy od k). W algorytmie drugim przeszukujemy wszystkie elementy w poszukiwaniu najdłuższego podciągu, więc czas przeszukiwania jest proporcjonalny do **n**.

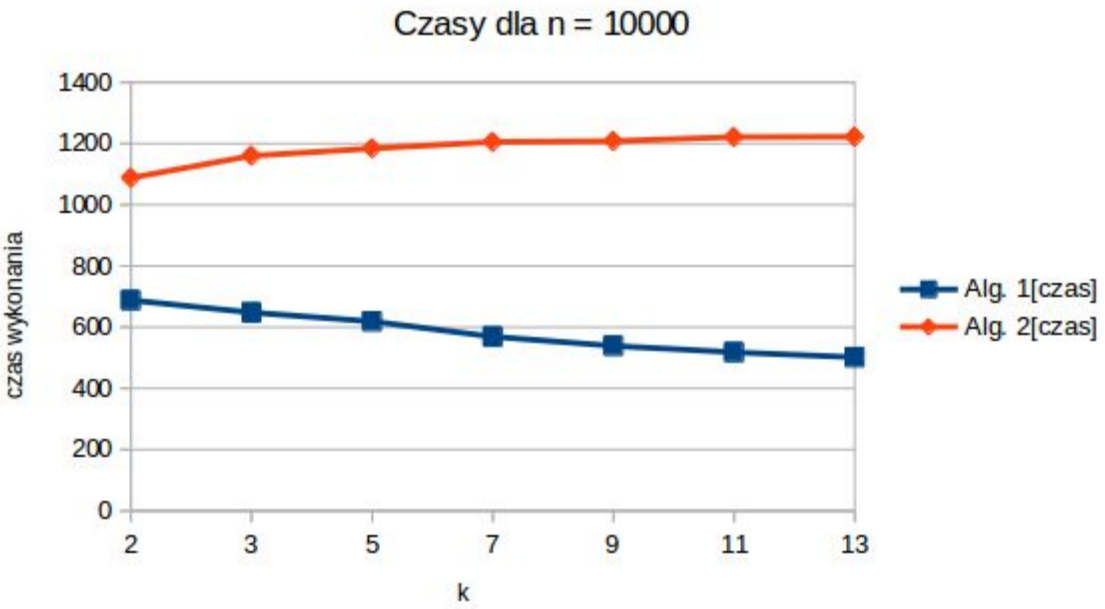
2. Korygujemy pozycję elementu
   W tym etapie ustawiamy element tak, by między jego pozycją, a pozycją bieżącego elementu było k elementów. W typowym przypadku wystarczy do tego jeden, lub zero ruchów. Natomiast nawet w najgorszym przypadku nie będzie to więcej niż **k** ruchów. Więc czas wykonania nie jest zależny od **n**.
3. Przestawiamy element na bieżącą pozycję
   W najgorszym przypadku musimy wykonać (n-bieżącaPozycja)/k przestawień, stąd czas wykonania dla danej pozycji jest proporcjonalny do **n**.

Z tej analizy wynika, że złożoność jest iloczynem n (zewnętrzna pętla) i sumy składników stałych i liniowych (w wewnętrznej pętli), stąd złożoność wynosi $\mathbf{O(n^2)}$.

Porównanie algorytmów

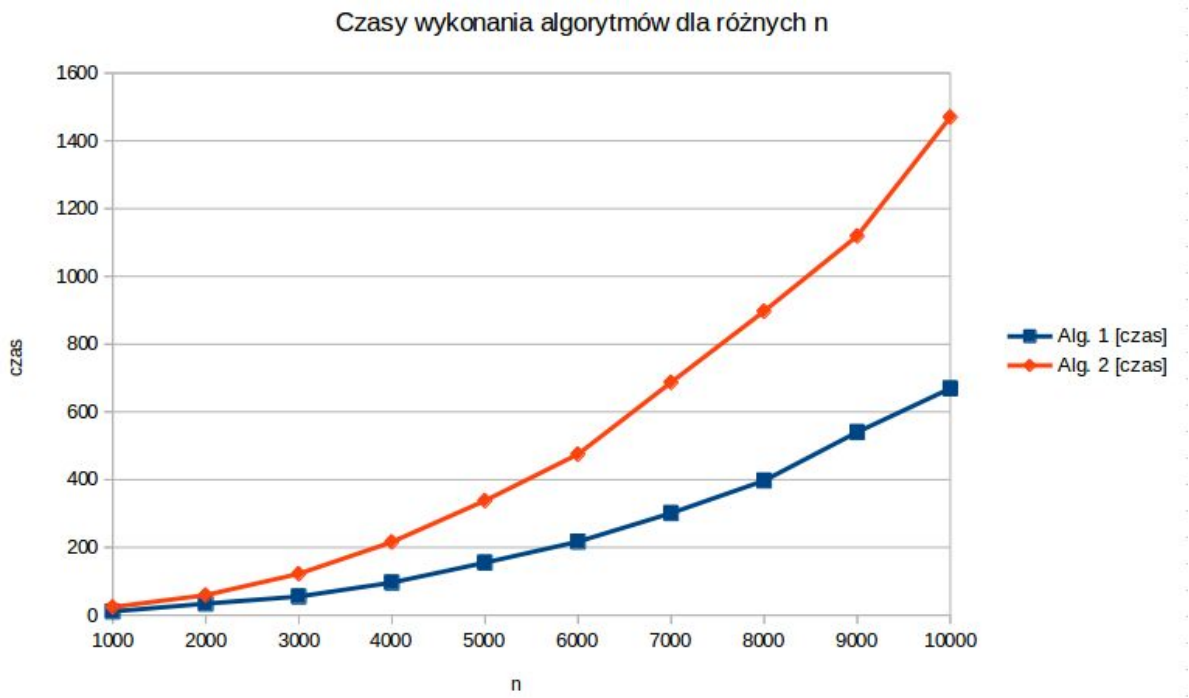


Na wykresach powyżej znajdują się wyniki przeprowadzonego przeze mnie eksperymentu polegającego na sprawdzaniu ilości przestawień prowadzących do posortowania linii produkcyjnej dla zmiennego k. Sprawdziłem jaką liczbę ruchów generuje każdy z dwóch algorytmów. Łatwo zauważyć, że dla małych k, algorytm drugi radzi sobie znacznie lepiej, wynika to z tego, że kiedy w skład kompletu wchodzi mała liczba elementów, to większe jest prawdopodobieństwo znalezienia
częściowo posortowanego kompletu. Pozwala to znacznie ograniczyć liczbę przestawień dla algorytmu drugiego, jednak dla większych k coraz mniejsze jest prawdopodobieństwo, że znajdziemy posortowany podciąg, więc traci on swoją przewagę.



Mimo przewagi i ilości posunięć, algorytm drugi pod względem czasu wykonania jest gorszy od pierwszego. Wynika to z faktu, że dla każdej kolejnej pozycji, na którą ma wstawić element, przeszukuje całą resztę linii produkcyjnej (by znaleźć największą posortowaną część kompletu). Natomiast algorytm pierwszy wybiera pierwszy znaleziony pasujący element. Można też zauważyć, że dla większych k, czas wykonania rośnie, to wynika ze stopniowej utraty jego przewagi w ilości wykonywanych ruchów (omówionej wyżej).

Złożoność czasowa algorytmów



Na wykresie powyżej przedstawione są wyniki eksperymentu polegającego na mierzeniu czasów wykonania programu dla zmiennych wartości **n**, (gdy k = 7). Można zauważyć, że wykres ma kształt paraboliczny, co wynika z kwadratowej złożoności algorytmu.

W sprawdzeniu poprawności oszacowanej złożoności wykonany został jeszcze jeden eksperyment. Polegał on także na pomiarach czasu dla stałego k (równego trzy). W tabelach poniżej znajdują się wyniki eksperymentu dla kolejno algorytmu pierwszego i drugiego.

Przy czym:
n - wielkość problemu
t(n)[ms] - czas rozwiązania w milisekundach

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

T(n) oznacza oszacowaną złożoność algorytmu, w tych przypadkach jest to n^2. Utrzymywanie się wartości q(n) w okolicach 1 oznacza dobre oszacowanie złożoności.

n	t(n)[ms]	q(n)
3000	63	1.02857
3300	70	0.94451
3600	87	0.986395
3900	99	0.956406
4200	123	1.02457
4500	138	1.00136
4800	157	1.00128
5100	177	0.999929
5400	199	1.00277
5700	216	0.976878
6000	245	1
6300	279	1.0329
6600	300	1.01198
6900	324	0.999961
7200	364	1.03175
7500	383	1.00049
7800	430	1.03852
8100	458	1.02573
8400	482	1.00375
8700	532	1.03278
9000	570	1.03401

n median: 6000 t(n) median: 245

n	t(n)[ms]	q(n)
3000	108	0.997691
3300	122	0.931422
3600	147	0.943033
3900	181	0.989382
4200	211	0.994486
4500	230	0.944316
4800	267	0.963482
5100	308	0.984521
5400	348	0.992216
5700	385	0.985203
6000	433	1
6300	465	0.974062
6600	517	0.986773
6900	561	0.979669
7200	633	1.0152
7500	668	0.987344
7800	731	0.998948
8100	797	1.00996
8400	852	1.00391
8700	908	0.997383
9000	988	1.01411

n median: 6000 t(n) median: 433