Lekcja 3: Algorytmy sortowania i przeszukiwania

Wstęp

Zajmiemy się teraz sortowaniem i wyszukiwaniem. Na razie z użyciem struktur, które dobrze znacie - tablic. I często z wykorzystaniem dopiero co poznanej rekurencji. Będziemy tablice sortować, będziemy w nich poszukiwać danych. Przy użyciu tablic będziemy poszukiwać wzorca w tekście.

Elementarne algorytmy sortowania

Większość projektów komputerowych nie mogłoby funkcjonować bez użycia algorytmów sortowania. Sortowanie złożonych zbiorów danych ułatwia późniejsze ich wykorzystanie w programie. W przypadku, gdy jest potrzebne posortowanie stosunkowo niewielkiej ilości danych, do tego celu wystarczy użycie prostych, elementarnych metod sortowania, które zostaną omówione w pierwszej części tego rozdziału. Metody sortowania przez wybieranie, wstawianie czy zamianę charakteryzują się złożonością obliczeniową $O(n^2)$, zatem w przypadku dużych wartości n użycie takich algorytmów znacznie obniża szybkość obliczeń i wskazane jest w tym wypadku wybrać algorytm bardziej złożony, ale charakteryzujący się większą efektywnością.

Warto sformalizować w tym miejscu nasz problem. Otóż problem sortowania jesteśmy w stanie zapisać tak:

- mamy zbiór danych zapisanych zebranych w pewnej strukturze przechowującej dane (w naszym przypadku będziemy używali tablicy, innym rozwiązaniem jest lista, która będzie omówiona w dalszych rozdziałach)
- oznaczmy tablicę danych jako T
- ilość elementów do posortowania wynosi n (znajdują się one w tablicy T, powiedzmy od T[0] do T[n-1])
- zadanie sortowania polega na znalezieniu takiej kolejności zapisu danych w tablicy (czyli permutacji zbioru n wartości), przy której
$$\forall_{0 \le i < n-1} \; T[i] \le T[i+1]$$
- jeśli powyższa relacja będzie spełniona, to dane zostaną posortowane w kolejności niemalejącej.

Jaki typ danych może być sortowany? Praktycznie każdy, na którym może określić relację porządku. Mogą to być dane zawierające liczby całkowite, znaki lub też rekordy – w przypadku których sortowanie następuję po jednym z ich pól.

Bardzo ważny jest podział technik sortowania ze względu na miejsce zapisu danych. Mianowicie dane mogą być przechowywane:

- w pamięci operacyjnej komputera – wtedy mamy do czynienia z sortowaniem wewnętrznym, a dane są najczęściej zapisane w strukturze tablicowej
- w plikach znajdujących się na dysku (czyli de facto w pamięci zewnętrznej), w których dane są równie często zapisywane w formie list, a do ich posortowania używa się metod sortowania zewnętrznego

W tej lekcji zajmiemy się wyłącznie operacjami sortowania danych umieszczonych w tablicach. W następnej lekcji poznacie bardziej zaawansowane struktury danych, i wtedy do sortowania jeszcze powrócimy.

Pierwszą grupa danych, którą omówimy w tym rozdziale, są metody sortowania elementarnego, do których należą przede wszystkim:

- sortowanie **przez wybieranie** – *SelectionSort*
- sortowanie **przez wstawianie** – *InsertionSort*
- sortowanie **przez zamianę** (tzw. bąbelkowe) – *BubbleSort*

SelectionSort

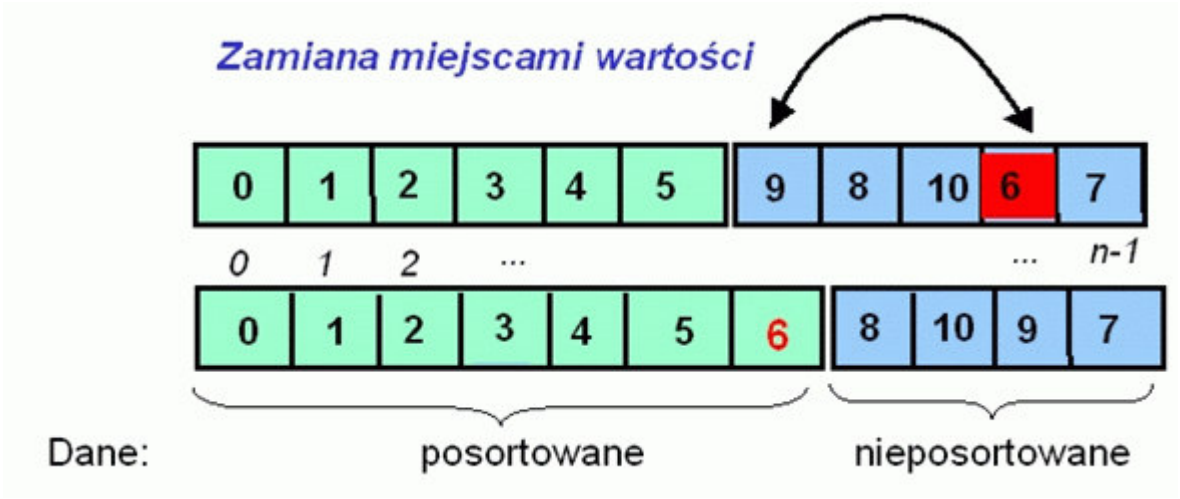
Jest to prosty i zarazem klasyczny rodzaj sortowania. Zakładamy, że mamy pewien zbiór danych o liczności n. Spośród n danych wybieramy wartość najmniejszą i wstawiamy na początek zbioru posortowanego. W tym momencie dysponujemy już zbiorem (n-1) danych nieposortowanych, z którego znów selekcjonujemy tę daną o najmniejszej wartości i przenosimy na drugie miejsce do zbioru posortowanego. Czynność tę wykonujemy aż do momentu, gdy zbiór nieposortowany (zawierający początkowo wszystkie dane) stanie się zbiorem pustym.

Pseudokod algorytmu *SelectionSort* jest bardzo przejrzysty:

Procedura **SelectionSort** (tablica **T**)

```
1. // n - liczba  elementów tablicy  T, o indeksach od 0 do n-1
2. for (indeks i = 0 ; T[i] nie jest ostatnim elementem T; zwiększaj i o 1) {
3.     znajdź indeks komórki z najmniejszym  elementem (powiedzmy indeksMin)
4.     spośród T[i]..T[n-1]
5.     zamień wartości komórek T[i] i T[indeksMin]
6. }
```

Wstawienie na odpowiednie miejsce siódmego z kolei najmniejszego elementu (7. krok iteracji) jest przedstawiony na poniższym schemacie:



Sortowanie przez wybieranie zawsze wykonuje n-1 zamian wartości komórek (przestawień), natomiast liczba porównań wartości komórek odpowiada złożoności obliczeniowej $O(n^2)$, a zatem sumarycznie całkowity czas sortowania jest rzędu $O(n^2)$, co przy małej liczbie danych umożliwia efektywne wykorzystanie *SelectionSort*.

InsertionSort

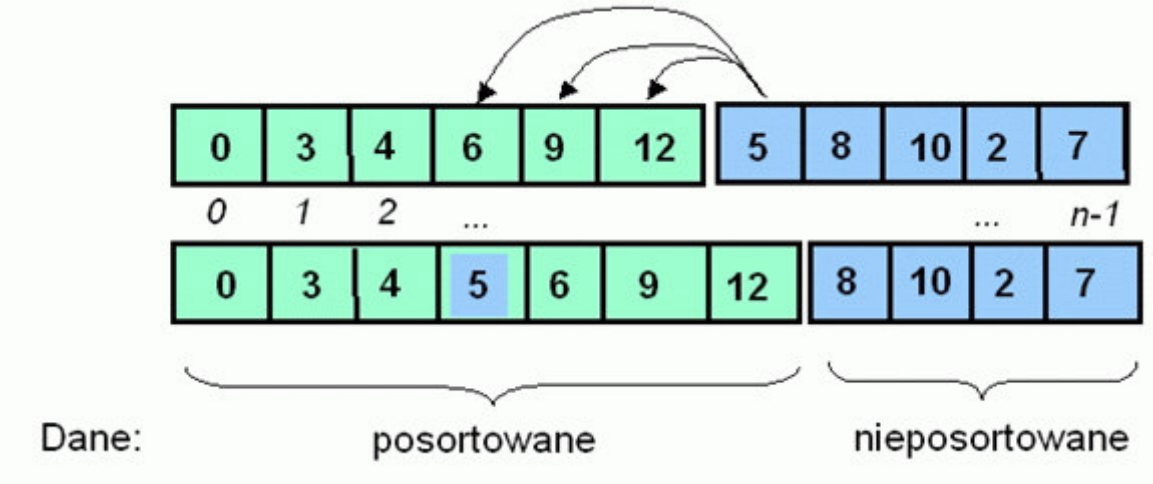
Drugim algorytmem sortowania elementarnego jest sortowanie przez wstawianie. Metoda ta jest nawet częściej używana w naszym życiu codziennym niż sortowanie przez wybór – przykładowo stosujemy ją do sortowania kart do gry znajdujących się w naszym reku. Zastanówmy się jak to czynimy – powiedzmy, że w lewej ręce trzymamy karty już posortowane (zaczynamy od umieszczenia w niej pierwszej karty), następnie bierzemy drugą kartę z części nieposortowanej i wybieramy dla niej miejsce w ciągu kart posortowanych w taki sposób, aby powiększony zbiór kart nadal posiadał cechę zbioru posortowanego. Kontynuujemy wstawianie kolejnych kart aż do momentu, gdy wszystkie karty zostaną umieszczone w lewej ręce.

W celu przybliżenia tej metody sortowania warto zaprezentować jej algorytm zapisany w pseudokodzie:

Procedura **InsertionSort** (tablica **T**)

```
1.  // n - liczba  elementów tablicy  T, o indeksach od 0 do n-1
2.  for (indeks j=1; j <=n-1; zwiększaj j o 1) {
3.      zapamiętaj wartość temp = T[j]
4.      indeks i = j -1
5.      while (i>= 0 oraz T[i]> temp) {
6.          T[i+1] = T[i] // przesuwamy w prawo element większy od temp
7.          zmniejsz i o 1
8.      }
9.      T[i+1] = temp // znaleziono miejsce dla nowego elementu z nieposortowanej części
10. }
```

Poniżej przedstawiamy w sposób graficzny jeden krok algorytmu – gdy próbujemy znaleźć właściwe miejsce dla elementu o wartości 5.



Tak jak w przypadku metody *SelectionSort*, sortowanie przez wstawianie posiada złożoność obliczeniową $O(n^2)$, a zatem użycie tego algorytmu jest zalecane jedynie przy małej liczbie danych - gdyż szybkość obliczeń będzie niewiele mniejsza niż w przypadku algorytmów bardziej złożonych, a na korzyść funkcji *InsertionSort* przemawia prostota i przejrzystość jej zapisu.

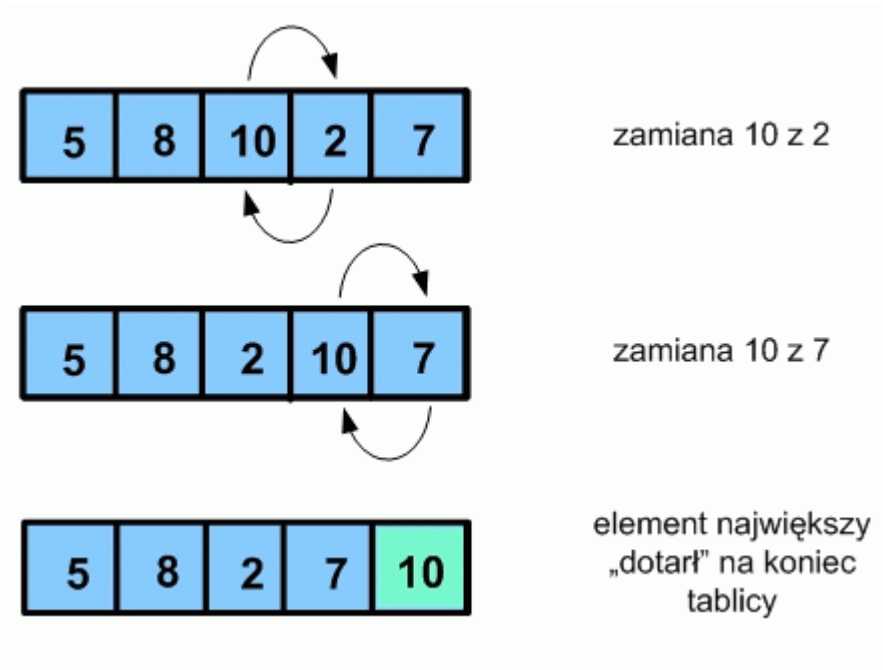
BubbleSort

Sortowanie bąbelkowe (czasem spotykane jest inne określenie – sortowanie przez zamianę) jest trzecim algorytmem porządkowania danych należącym do metod sortowania elementarnego. Nazwa algorytmu pomaga w ogólnym zrozumieniu jego działania - w podobny sposób, jak bąbelki powietrza unoszą się w kierunku powierzchni wody, tak samo również elementy większe od pozostałych są „przesuwane" w prawą stronę tablicy T. Owo przemieszczanie się elementów polega na zamianie miejscami danych zapisanych w sąsiednich komórkach (w przypadku gdy nie spełniają one relacji porządku), poczynając od porównania T[0] i T[1], następnie T[1] i T[2] aż do ostatniej pary komórek tablicy. Algorytm wykonuje się do momentu, aż cały ciąg danych będzie posortowany. W pierwszym kroku iteracji zostanie przesunięty na koniec tablicy element największy (do komórki T[n-1]), w kolejnym - element posiadający drugą wartość co do wielkości (do komórki T[n-2]), w następnym – trzecią wartość itd..

Procedura **BubbleSort** (tablica **T**)

```
1. // n - liczba  elementów tablicy  T, o indeksach od 0 do n-1
2. for (indeks i=0; i<n-1; zwiększaj i o 1) {
3.     for (indeks j=0; j<=n-2-i; zwiększaj j o 1) {
4.         if (T[j] > T[j+1]) {
5.             zamień miejscami T[j] i T[j+1]
6.         }
7.     }
8. }
```

W metodzie bąbelkowej występuje duża liczba zamian danych i nie mniejsza liczba porównań. Złożoność obliczeniowa BubbleSort jest, podobnie jak wszystkie elementarne algorytmy sortowania, $O(n^2)$. Ilość zamian praktycznie uniemożliwia jej zastosowanie w przypadku operowania na większych zbiorach danych.



Porównanie efektywności elementarnych metod sortowania

Wszystkie trzy omówione algorytmy charakteryzują się złożonością obliczeniową $O(n^2)$ – z tego powodu programiści korzystają z nich tylko przy porządkowaniu małych zbiorów danych. Sortowanie bąbelkowe przebiega najwolniej z całej grupy (średnio 3 razy wolnej niż sortowanie przez wstawianie) - podyktowane jest to zdecydowanie największą liczbą zamian elementów („unoszących się pęcherzyków"). *BubbleSort* jest bardziej przydatny niż sortowanie przez wybór czy sortowanie przez wstawianie jedynie w przypadku danych prawie posortowanych.

Jeślibyśmy chcieli porównać czasy sortowania przez wstawianie i wybieranie, to okaże się, że algorytm *InsertionSort* jest na ogół skuteczniejszy niż *SelectionSort* – dla losowych danych wykonuje się on około 1,5 raza szybciej. Dotyczy to szczególnie przypadku, gdy porządkujemy elementy, które (pojedynczo) zajmują mało miejsca w pamięci (np. klucze rekordów będące liczbami całkowitymi), a także gdy dane są prawie posortowane. Natomiast w przypadku sortowania rekordów od dużym rozmiarze (kluczy wraz z danymi) – metoda *SelectionSort* okazuje się dużo efektywniejsza od sortowania przez wstawianie. Wynika to z faktu stałej, niewielkiej liczby zamian (równej n-1) w sortowaniu przez wybieranie (a wiemy, że przenoszenie rekordów o dużych rozmiarach jest czasochłonne). Niestety, żadna modyfikacja wyżej wymienionych algorytmów nie będzie miała znaczenia, jeśli przed nami stanie zadanie posortowania bardzo dużej liczby elementów. Do tego celu przeznaczone są algorytmy bardziej złożone, o których dowiemy się już niebawem.

Sortowanie szybkie – QuickSort

Algorytm sortowania szybkiego jest dziś prawdopodobnie najczęściej stosowaną metodą porządkowania danych. Został on opracowany przez Hoare'a niemalże pół wieku temu, bo w roku 1960. Jego dużą zaletą jest złożoność obliczeniowa wynosząca dla średniego przypadku **O(n lg n)**, mimo że przy pesymistycznym ułożeniu danych algorytm wykonuje się w czasie $O(n^2)$, podobnie jak metody sortowania elementarnego. *QuickSort* jest przede wszystkim używany do porządkowania dużych zbiorów danych, natomiast jest on niewart zastosowania, gdy liczba elementów do sortowania jest niewielka – efektywniejsze są wówczas algorytmy elementarne (takie jak sortowanie przez wybieranie).

Hoare jako zalążek nowego algorytmu wybrał metodę *BubbleSort* – ze względu na przypuszczalne możliwe ulepszenia. Odkrył on, że rozsądniejszym ruchem niż zamiana sąsiednich elementów może okazać się wymiana danych znajdujących się daleko od siebie. Autor algorytmu *QuickSort* doszedł do wniosku, że zastosowanie takiej idei może w pokaźny sposób ograniczyć ilość przestawień danych.

Sortowanie szybkie jest metodą opartą na zasadzie „dziel i zwyciężaj". W dalszym ciągu ustalamy, że nasze dane są przechowywane w tablicy T. Dla ogólności metody podziału przyjmujemy, że operujemy na podtablicy T[a..c]; w tym momencie interesują nas komórki od T[a] do T[c] – w pierwszym kroku algorytmu a=0, c=n-1, gdzie n określa rozmiar tablicy T:

- Etap **„dziel"**:
  - Wybieramy jeden z elementów T[a..c] (później przedstawimy różne reguły jego wyboru) – będzie on określany jako **element rozdzielający** bądź **osiowy**.
  - Algorytm dzieli tablicę T[a..c] na dwie podtablice - stosując zamiany elementów i ustalając indeks b - w taki sposób, że w podtablicy T[a..b-1] znajdują się elementy nie większe niż wartość elementu osiowego, a w

podtablicy T[b+1..c] są tylko elementy nie mniejsze niż wartość elementu osiowego.
    - Element rozdzielający zostaje umieszczony w komórce T[b].
- Etap **„zwyciężaj”**:
    - Porządkowanie obu podtablic T[a..b-1] oraz T[b+1..c] opiera się na zasadzie rekurencyjnych wywołań metody QuickSort, jako argumenty przyjmując właśnie nowe podtablice.
    - Proces ten jest powtarzany aż do momentu, gdy cała tablica T zostanie podzielona na podtablice składające się z jednego elementu - taka sytuacja nie wymaga już dalszego sortowania.
- Etap **„połącz”**:
    - Wszystkie przestawienia elementów są wykonywane na oryginalnej tablicy T, więc w tym momencie możemy stwierdzić, że tablica T została posortowana.

Zastanówmy się, na jakiej zasadzie należałoby wybierać element rozdzielający. Optymalnym doborem byłoby wybranie **mediany**, czyli elementu, względem którego znajduje się tyle samo danych nie większych i nie mniejszych w tablicy T. Jednak samo znalezienie mediany przy dużych rozmiarach zbioru danych wymaga efektywnego algorytmu i pochłania sporo czasu obliczeniowego.

Dlatego też najczęściej na element rozdzielający wybiera się pierwszy, środkowy bądź ostatni element tablicy T. Wersje z użyciem elementu pierwszego bądź ostatniego są, można powiedzieć, symetryczne, więc postaramy się jedynie porównać zasady podziału podtablic korzystające z ich elementu środkowego lub ostatniego jako elementu osiowego.

Algorytm etapu „dzielenia” jest najczęściej umieszczany w osobnej funkcji, która jest wywoływana z poziomu metody *QuickSort*. Poniżej przedstawiamy pseudokod takiej funkcji w wersji z ostatnim elementem podtablicy jako elementem rozdzielającym:

Funkcja **Partition** (tablica **T**,indeks **a**, indeks **c**) typu całkowitego

```
1.  oś = T[c]
2.  i = a-1
3.  for (j=a; j jest mniejsze od c; zwiększaj j o 1) {
4.      if (T[j]<=oś) {
5.          zwiększ i o 1
6.          zamień T[i] z T[j]
7.      }
8.  }
9.  zamień T[i+1] z T[c]
10. return wartość i+1
```

Wiemy już, jak przestawia się algorytm dzielenia podtablic. Okazuje się, że kod metody *QuickSort*, zgodny z treścią etapu „zwyciężaj”, jest wyjątkowo prosty. Zatem właściwa metoda sortująca polega „jedynie” na rekurencyjnym wywoływaniu samej siebie oraz wywoływaniu funkcji Partition, która de facto odpowiada za przebieg porządkowania danych.

Procedura **QuickSort** (tablica **T**,indeks **a**, indeks **c**)

```
1.  b = Partition(T,a,c)
2.  if (b-1>a)        // nie dzielimy podtablic jednoelementowych
3.      QuickSort(T,a,b-1)                                  // rekurencyjne wywołanie
4.  if (b+1<c)      // jak wyżej
5.      QuickSort(T,b+1,c)                                // rekurencyjne wywołanie
```

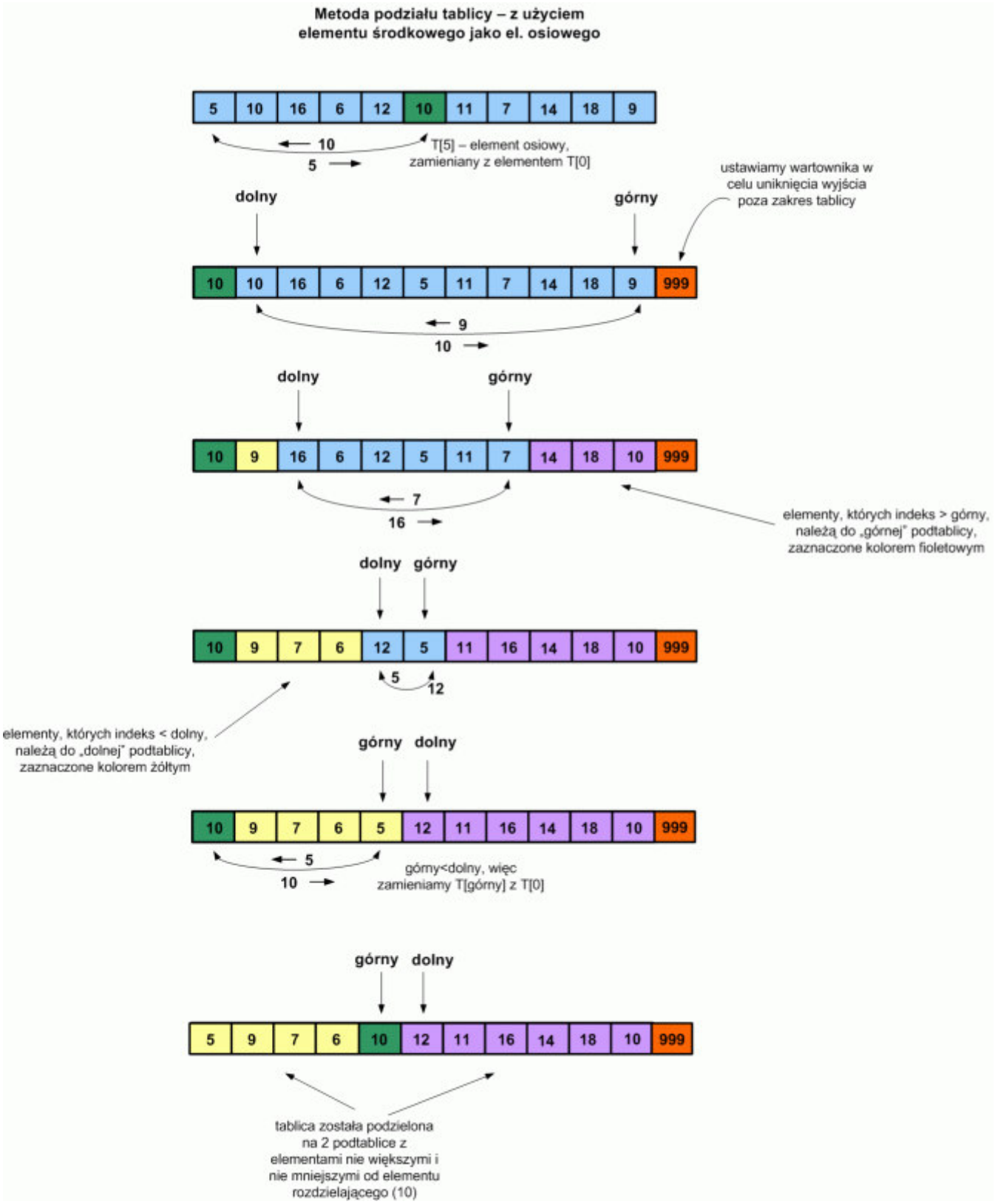
Co zrozumiałe, w celu posortowania całej tablicy T wywołujemy funkcję *QuickSort* z argumentami a = 0, c = n-1.

Jak już wcześniej wspomnieliśmy, ciekawą zagadnieniem będzie porównanie kroku dzielenia podtablicy za pomocą elementu osiowego, który:

- jest środkowym elementem podtablicy T[a..c]
- jest ostatnim elementem podtablicy T[a..c]

Niżej zostanie zaprezentowany w sposób graficzny jeden etap typu „dziel” wykonujący się na tej samej podtablicy dla obu wspomnianych wariantów wyboru elementu rozdzielającego.

W pierwszej kolejności pokażemy wykonanie funkcji Partition w wersji wyboru środkowego elementu tablicy jako elementu rozdzielającego:



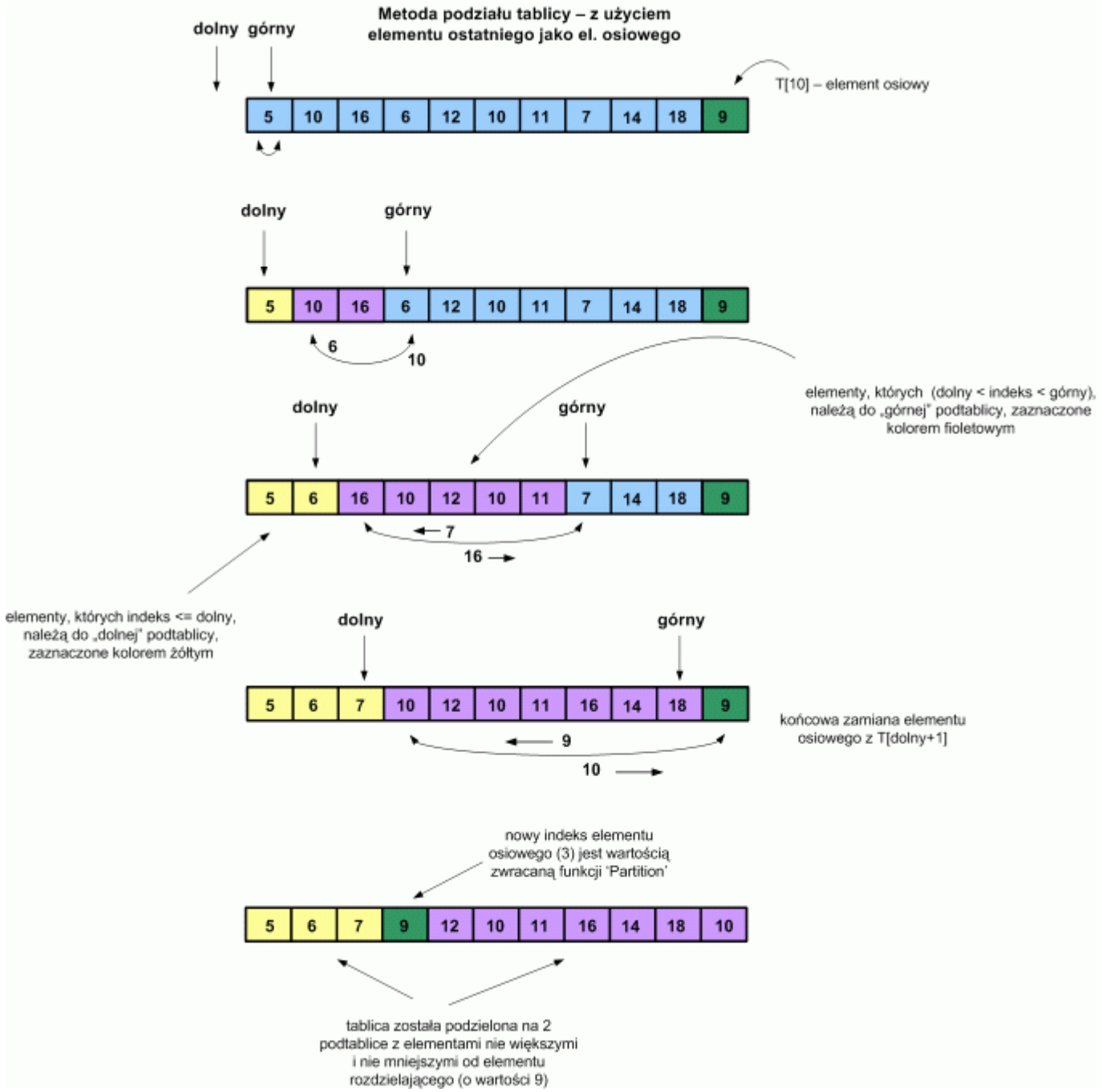
W skrócie metoda podziału w tej wersji wykonuje się w następujących krokach:

- zamieniamy środkowy element tablicy z pierwszym elementem
- ustawiamy początkowe położenie „wskaźników": dolny na T[1], górny na T[n-1]
- w dodatkowej komórce T[n] umieszczamy wartownika o bardzo dużej wartości
- dolny i górny zaczynają się przesuwać, dolny „w prawo", górny „w lewo"
    - w momencie, gdy T[dolny] >= element osiowy, wskaźnik zatrzymuje się
    - wskaźnik górny podobnie się zatrzymuje, gdy T[górny] <= element osiowy
    - następuje zamiana elementów T[dolny] oraz T[górny]
    - wskaźniki przemieszczają się dalej aż do następnej zamiany
    - algorytm kończy się, gdy wskaźniki „miną się", tzn. dolny > górny
- na koniec należy zamienić T[0] z T[górny] – element osiowy znajdzie się we właściwym miejscu

Warto jeszcze wspomnieć o funkcji wartownika. Wskaźnik dolny i górny przesuwają się w odpowiednie strony. Wskaźnik górny, przesuwając się w lewo, w najgorszym przypadku zatrzyma się na elemencie osiowym, natomiast w wskaźnik dolny w momencie przesuwania się w prawo mogły w pesymistycznym wariancie przejść przez całą tablicę i wyjść poza jej zakres – mamy

świadomość, że byłoby to niebezpieczne i niekontrolowane zjawisko. Dlatego też powyższy algorytm zakłada umieszczenie wartownika „na skraju" tablicy T, którego duża wartość zabezpiecza wskaźnik „dolny" przed wyjściem poza tablicę.

Poniżej zamieszczamy w formie schematu rysunkowego metodę dzielenia tablicy w wersji, którą już wcześniej poznaliśmy – gdzie ostatni element tablicy jest elementem osiowym. Pozostajemy przy tej samej tablicy wejściowej:



Na koniec rozważań dotyczących zastosowania sortowania szybkiego wskazane jest poświęcenie chwili uwagi złożoności obliczeniowej omawianego algorytmu. Okazuje się bowiem, że jest spora różnica czasowa przy sortowaniu danych w optymistycznej i pesymistycznej sytuacji. Algorytm najdłużej się wykonuje się okoliczności, gdy elementem rozdzielającym za każdym razem okazuje się element podtablicy T o skrajnej wartości (maksymalnej lub minimalnej). W takim przypadku podtablica nie jest praktycznie dzielona, lecz „odpada" od niej jedynie element osiowy, a pozostałą część podtablicy należy dzielić dalej.

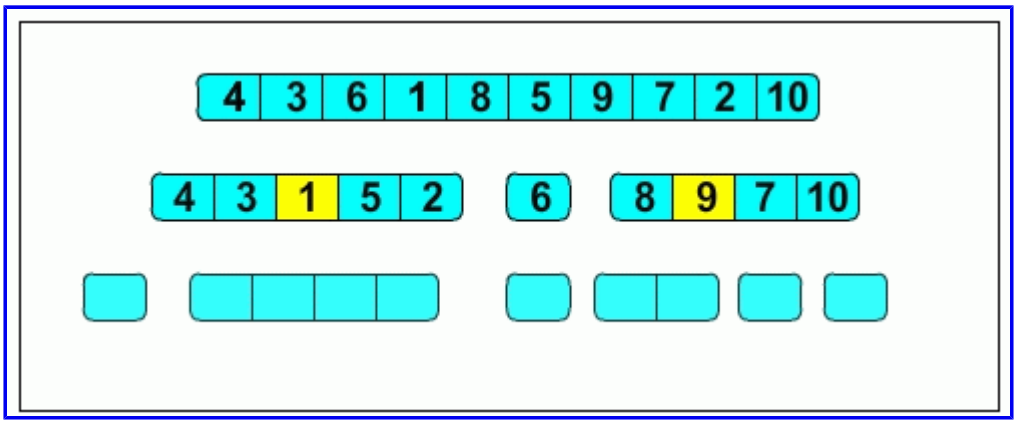
Złożoność obliczeniowa pesymistycznego układu danych wynosi aż $O(n^2)$. Należy sobie zadać pytanie, jakie początkowe ułożenie danych zredukowałoby czas sortowania do minimum. Intuicyjnie wydaje nam się, że „optymalne" dane wejściowe powodują każdorazowe wybieranie elementu osiowego, który dzieli podtablice równo bądź prawie na pół. W takiej sytuacji łączny czas obliczeń sortowania sposobem *QuickSort* jest rzędu O(n lg n).

Wiemy jednak, że przypadek optymistyczny, jak i pesymistyczny statystycznie zdarza się niezmiernie rzadko. Bardziej interesujące jest, jaka jest złożoność algorytmu dla typowego ułożenia danych. Wyniki wielu takich testów są zadowalające - wynika z nich bowiem, że nawet mimo względnie „pechowego" ułożenia danych złożoność obliczeniowa uporządkowania ich sortowaniem szybkim nadal wynosi O(n lg n).

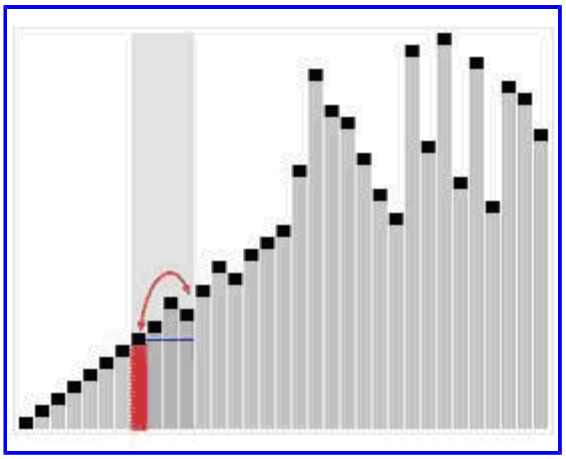
Od momentu powstania pierwszej wersji *QuickSort* powstało już bardzo wiele wariantów tego algorytmu, które różnią się między

sobą głównie zasadą doboru elementu osiowego. Wyszukując informacji o sortowaniu szybkim w różnych książkach i materiałach w Internecie, bądźmy czujni i obserwujmy, jak wybierany jest element osiowy (i niech nas nie dziwi fakt, że jest niemalże tyle odmian algorytmu *QuickSort*, ile jest podręczników). W tym rozdziale wspomnieliśmy już o wyborze pierwszego, ostatniego bądź środkowego ze zbioru elementów – inne proponowane metody to wybór mediany spośród wymienionych przed chwilą elementów, a także losowy dobór elementu rozdzielającego. Liczba wersji metody sortowania szybkiego nie powinna dziwić – wszak jest on uniwersalnie stosowany w wielu różnorodnych sytuacjach, które zachęcają do jeszcze skuteczniejszego ulepszania algorytmu.

Interesującym zagadnieniem wydaje się też analiza nierekurencyjnego zapisu algorytmu sortowania szybkiego, w którym stos jest używamy w sposób jawny. Wszystkich zainteresowanych odsyłamy do uniwersyteckiego portalu informatycznego **WAŻNIAK** pod adresem http://wazniak.mimuw.edu.pl. Przy okazji warto zobaczyć animację prezentującą przykładowe wykonanie algorytmu *QuickSort*; jest ona na końcu strony, która otworzy się po kliknięciu w poniższy obrazek będący zrzutem ekranu z tej animacji:



Polecamy również obejrzenie innej ciekawej animacji przedstawiającej sortowanie szybkie - na stronie Wikipedii:



Sortowanie przez scalanie - MergeSort

Zbliżonym algorytmem do metody *QuickSort* jest sortowanie przez scalanie (ang. *MergeSort*). Jego idea została przedstawiona po raz pierwszy przez Johna von Neumanna, jednego z pionierów informatyki. Koncepcja sortowania przez scalanie jest przejrzysta. Metoda *MergeSort* opiera się, podobnie jak algorytm sortowania szybkiego, na zasadzie „dziel i zwyciężaj". W tym przypadku kroki tej reguły można wyrazić następująco:

- **„dziel"** - załóżmy, że dane przechowujemy w tablicy o rozmiarze n. Dzielimy ją na dwie podtablice o rozmiarze n/2 (czyli na połowy)
- **„zwyciężaj"** – sortujemy nowo powstałe tablice, wywołując rekurencyjnie algorytm *MergeSort* aż do momentu, gdy nowe podtablice będą jednoelementowe
- **„połącz"** – scalamy dwie podtablice w jedną posortowaną tablicę

Jaka jest zasadnicza różnica między algorytmami *QuickSort* i *MergeSort*? Wydaje się, że przed wszystkim uproszczony algorytm podziału tablic na części w przypadku sortowania przez scalanie. W metodzie sortowania szybkiego podstawowym problemem było takie dobieranie elementu rozdzielającego, aby podział na podtablice był jak najbardziej równomierny (co powoduje, że w pesymistycznym przypadku algorytm *QuickSort* wykonuje się w czasie $O(n^2)$ – za sprawą skrajnie niekorzystnych podziałów tablic).

Natomiast algorytm podziału tablicy w *MergeSort* jest uproszczony do minimum – zawsze dzielimy ją na połowy, nie wnikając w konfigurację danych w tablicy. Cały mechanizm sortowania jest osadzony w części algorytmu zajmującej się scalaniem 2 podtablic. Można powiedzieć, że jest tutaj odwrotna sytuacja niż w przypadku sortowania szybkiego. W metodzie sortowania przez scalanie zasadnicze porządkowanie danych następuje w momencie łączenia tablic, a dzielenie ich jest uproszczone. Z kolei

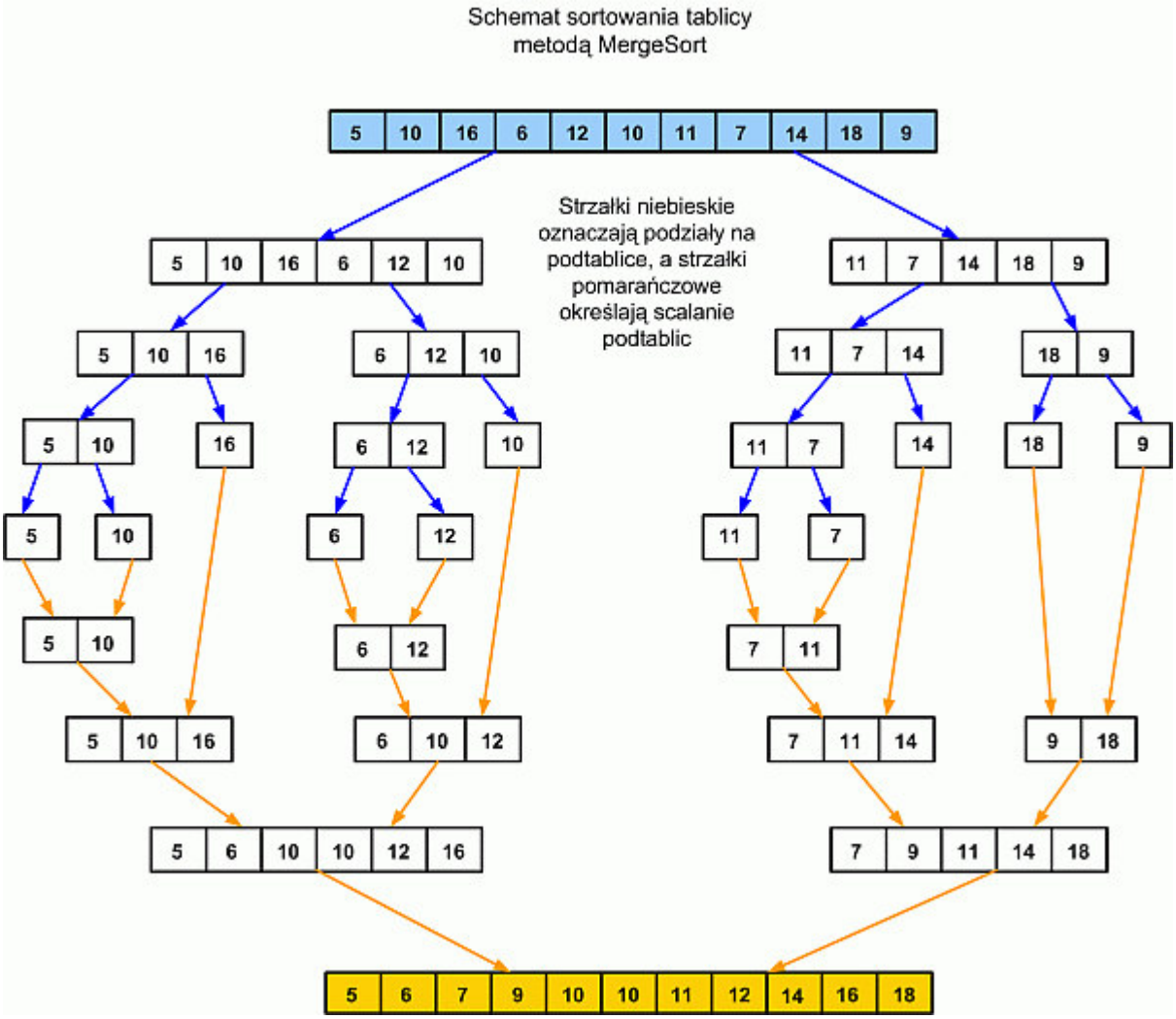
w metodzie *QuickSort* to właśnie podczas podziału tablic następuje sortowanie danych, a łączenie podtablic jest „mechaniczne”.

Przedstawione powyżej podstawowe zasady algorytmu sortowania przez scalanie można przedstawić w formie pseudokodu:

Procedura **MergeSort** (tablica **T**)

```
1. if  (T zawiera więcej niż 1 element) {
2.     podziel T na dwie połowy (podtablice)
3.     MergeSort(„lewa” podtablica T)          // rekurencyjne wywołanie
4.     MergeSort(„prawa” podtablica T)        // rekurencyjne wywołanie
5.     scal obie podtablice w posortowaną tablicę T
6. }
```

Zasadę działania sortowania ze scalaniem powinien wyjaśnić poniższy rysunek:



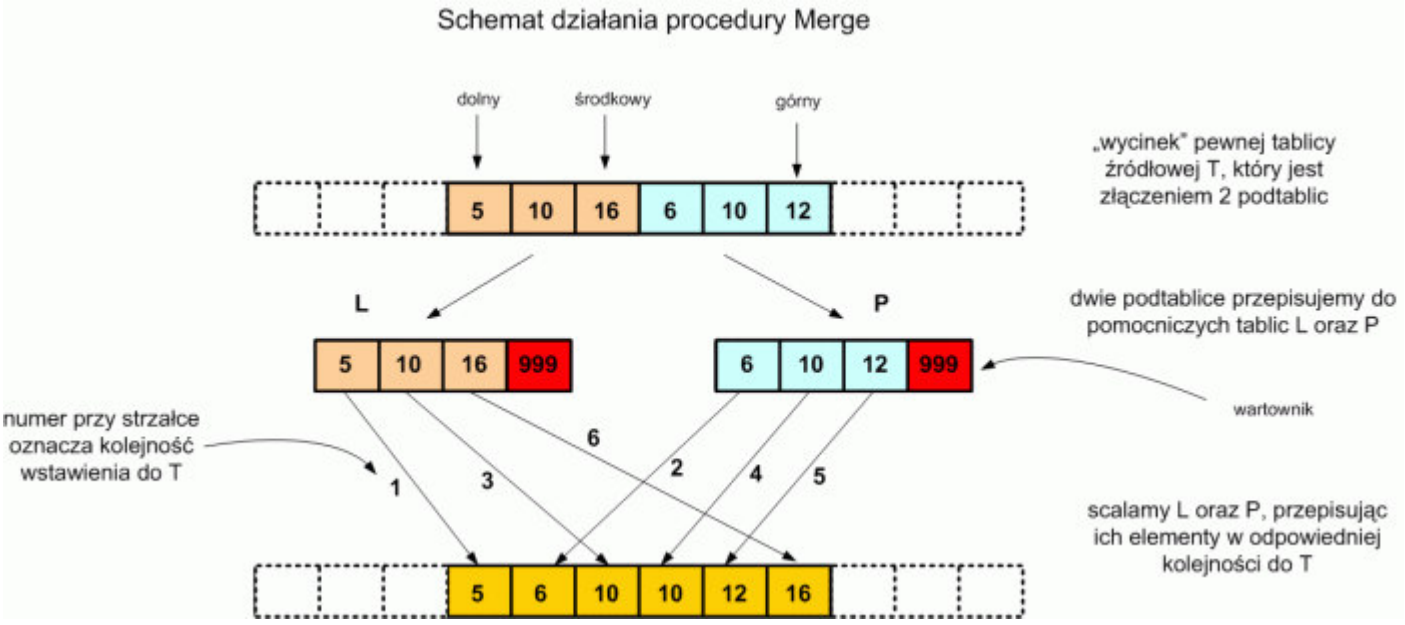
Scalenie podtablic odbywa się poprzez użycie funkcji, która w implementacjach najczęściej nosi nazwę *Merge*. Jej zadaniem jest przenoszenie danych z dwóch posortowanych podtablic do tablicy w taki sposób, aby otrzymać ciąg danych posortowanych. W celu prostego wytłumaczenia zasady jej działania prezentujemy poniższy pseudokod funkcji *Merge*:

Procedura **Merge** (tablica **T**, indeks **dolny**, indeks **środkowy**, indeks **górny**)

```
1.  m1 =  środkowy - dolny + 1
2.  m2 =  górny - środkowy
3.  Utwórz tablicę L[0..m1]   //zawiera elementy lewej podtablicy
4.  Utwórz tablicę P[0..m2]  //zawiera elementy prawej podtablicy
5.  Przepisz T[dolny..środkowy] do L[0..m1-1]
6.  Przepisz T[środkowy..górny] do P[0..m2-1]
7.  Ustaw wartowników w L[m1] i P[m2]
8.
9.  i=0,   j=0
10. for (k=dolny; k<=górny; zwiększaj k o 1) {
11.     if  (L[i] <= P[j]) {
12.         T[k] = L[i]
13.         zwiększ i o 1
14.     }
15.     else {
16.         T[k] = P[j]
17.         zwiększ j o 1
18.     }
19. }
```

Metoda *Merge* tworzy pomocnice tablice, do których przepisywane są dwie podtablice, które należy scalić. Do źródłowej tablicy T przepisywane są z powrotem elementy tablic L i P – za każdym razem porównujemy ze sobą 2 najmniejsze elementy L i P, które jeszcze nie zostały przepisane do T. Wybieramy mniejszego z nich, którego należy umieścić w „dużej" tablicy, a następnie procedurę porównywania i przekazywania elementów prowadzimy do momentu „wyczerpania się" tablic L oraz P. Aby zmusić algorytm do wykorzystania wszystkich elementów ze wspomnianych tablic, używamy wartowników o dużej wartości w ostatnich komórkach tablic – dzięki temu, gdy z jednej z tablic zostaną już przepisane wszystkie elementy, nastąpi potem przepisanie reszty pozostałych elementów w drugiej tablicy do tablicy T.

Być może nie jest jeszcze do końca zrozumiała zasada wykonywania procedury *Merge*, dlatego też warto przeanalizować niżej przedstawiony schemat:



Jak zwykle najważniejszym kryterium efektywności danego algorytmu sortowania jest jego złożoność obliczeniowa. Metoda *MergeSort*, podobnie jak należąca do tej samej klasy funkcja sortowania szybkiego, posiada złożoność obliczeniową O(n lg n) ze współczynnikiem 2 **(2n lg n)**. Jeślibyśmy chcieli porównać szybkość obliczeniową z metodami *QuickSort* (złożoność **1,4 n lg n)**) oraz *HeapSort* (o której dowiemy się przy okazji omawiania kopców – nastąpi to w dalszej części podręcznika), to okaże się, że dla losowych danych metoda *MergeSort* jest wolniejsza od sortowania szybkiego, ale szybsza niż sortowanie przez kopcowanie.

Do tej pory mało mówiliśmy o niedogodnościach tego algorytmu - otóż jest jedna zasadnicza wada jego używania. Mianowicie zauważyliśmy, że do scalania tablic potrzebna jest dodatkowo pomocnicza przestrzeń pamięci. Zatem może się okazać, że użycie algorytmu sortowania przez scalanie na bardzo dużych zbiorach danych będzie nie do zaakceptowania ze względu na konieczność wykorzystania zbyt dużej części pamięci komputera.

Na koniec warto wspomnieć o jeszcze innym, ważnym zastosowaniu algorytmu *MergeSort* - do sortowania zewnętrznego. Taki typ sortowania polega na porządkowaniu danych znajdujących się w różnych plikach, do których dostęp jest sekwencyjny (czyli nie jest natychmiastowy do wszystkich danych). Do realizacji takiej wersji *MergeSort* najwygodniejsze jest zastosowanie struktur listowych, które poznacie w następnym rozdziale. Wrócimy więc do sortowania przez scalanie w lekcji 4 na początku rozdzialu o sortowaniu kubełkowym.

Przeszukiwanie

Problem polegający na przeszukiwaniu (a raczej poszukiwaniu) pewnego przedmiotu lub informacji jest nam doskonale znany z życia codziennego. Praktycznie codziennie jesteśmy zmuszeni do znalezienia rzeczy nam niezbędnej. Czy to będzie koszula, czy może książka w domowej biblioteczce, nie ma znaczenia – chcemy, by się odnalazła w możliwie najkrótszym czasie.

Podobnie jest w informatyce. Zagadnienie przeszukiwania danych jest jednym z najbardziej podstawowych problemów, do rozwiązania których należy używać opracowanych algorytmów. W tym rozdziale przedstawimy najprostsze i zarazem najbardziej klasyczne metody wyszukiwania danych zapisanych w tablicach.

Dla uproszczenia modelu przeszukiwania przyjmiemy, że dane, wśród których będziemy „szukali" tej właściwej, będą znajdowały się w tablicy T[n], gdzie n oznacza rozmiar tablicy T i zarazem ilość danych. Oprócz tego posiadamy pewną wartość liczbową x (oczywiście ogólny problem polega na wyszukiwaniu danych dowolnego typu, choćby rekordów).

Zadanie, jakie jest przed nam postawione, można sformułować następująco:

- Określić, czy istnieje indeks i tablicy T taki, że T[i] = x.
- Jeśli tak, to algorytm musi nas o tym poinformować (np. zwrócić wartość i), natomiast jeśli taki indeks nie istnieje, tzn. nie ma wartości x w tablicy T, to również musimy się o tym dowiedzieć (np. algorytm zwróci wartość -1).

Przeszukiwanie liniowe

Najbardziej oczywistym i zarazem naturalnym sposobem wyszukiwania danych jest przeszukiwanie liniowe (sekwencyjne). Należy jednak na wstępie zaznaczyć, że używanie takiego rodzaju algorytmu jest godne polecenia jedynie w sytuacji, gdy nie posiadamy jakichkolwiek informacji o danych, które należy przeszukać.

Algorytm jest bardzo prosty. Odwołując się ponownie do aspektów życia codziennego, taki sposób przeszukiwania można porównać ze znajdywaniem książki na domowej półce. Nie są one w żaden sposób uporządkowane (alfabetycznie bądź tematycznie). W związku z tym nie pozostaje nam nic innego, jak przejrzeć po kolei, od lewej do prawej, wszystkie książki po kolei, chyba, że w pewnym momencie okaże się, że znaleźliśmy właśnie upragnioną książkę.

Jak się zapewne domyślacie, algorytm zapisany w języku programowania jest równie mało skomplikowany:

Funkcja **szukajLin** (tablica **T**, liczba **x**) typu całkowitego

```
1. // n  - rozmiar tablicy T
2. // i - indeks
3. for (i=0; i<n; zwiększaj i o 1)
4.     if (T[i] jest równe x)
5.         return i
6. return -1
```

W powyższym pseudokodzie informacja o nieznalezieniu elementu w tablicy jest przekazywana poprzez zwrócenie wartości –1 (jeśli funkcja *szukajLin* zwróci wartość z przedziału [0..n-1], oznaczać to będzie, że w jednej z komórek T jest zapisana wartość x.

Niestety, tak jak wcześniej to zakomunikowaliśmy, algorytm wyszukiwania liniowego jest bardzo wolny i przydatny wyłącznie przy przeszukiwaniu danych, na temat których nie posiadamy żadnej wartościowej wiedzy. Złożoność obliczeniowa algorytmów przeszukiwania liniowego zawsze będzie należeć do klasy **O(n)**.

Spójrzmy na problem wyszukiwania sekwencyjnego z innej strony. Załóżmy, że prawdopodobieństwo zdarzenia, że wartość x znajduje się w tablicy T (z jednakowym prawdopodobieństwem na którejkolwiek pozycji), wynosi p. Okazuje się, że wartość oczekiwana liczby porównań, które wykona algorytm *szukajLin*, wynosi **n*(1- p/2) +p/2**. Możemy z tej informacji wyciągnąć kilka interesujących konkluzji:

- jeśli p=1, to *średniaLPorównań(p,n)=(n+1)/2*
- jeśli p=0, to *średniaLPorównań(p,n)=n*

Bez trudu jesteśmy w stanie odnotować, że nawet jeśli jesteśmy pewni faktu, że poszukiwany element jest w zbiorze danych, to musimy średnio przejrzeć aż połowę elementów z tego zbioru. Wraz ze spadkiem wartości p liczba elementów zbioru, które należy zbadać i porównać z x, rośnie liniowo aż do n.

Przeszukiwanie binarne

W sytuacji, gdy dane są w pewien sposób uporządkowane (na przykład posortowane), znacznie bardziej użyteczna od wyszukiwania liniowego jest metoda przeszukiwania binarnego. Idea algorytmu jest taka, że dzielimy tablicę T na połowy i sprawdzamy wartość środkowej komórki tablicy (w ten sposób od razu pomijamy konieczność przeszukania połowy tablicy). Jeśli jej wartość jest tą, której szukamy, to wyszukiwanie jest zakończone. W przeciwnym razie dzielimy nową podtablicę („lewą"

bądź „prawą”, w zależności od wartości środkowej komórki tablicy i sposobu posortowania danych) i ponownie sprawdzamy wartość „nowej” środkowej komórki podtablicy – powtarzamy tę czynność, dopóki elementy środkowe podtablic nie są równe wartości szukanej oraz nowo powstała podtablica ma więcej niż jedną komórkę. W skrócie możemy przyjąć, że w kolejnych krokach algorytmu odrzucamy ½ tablicy T, potem ¼ itd.

Jest to klasyczny przykład algorytmu opierającego się na zasadzie „dziel i zwyciężaj” (przyjmijmy, że tablica T jest posortowana niemalejąco):

- w pierwszym kroku sprawdź, czy środkowy element tablicy T jest równy poszukiwanej wartości x; jeśli nie, to:
- „dziel” – podziel podtablicę(na początku właściwą) tablicy T na 2 podtablice (najczęściej na połowy) i przejdź do „lewej” podtablicy (gdy x jest mniejszy od środkowego elementu) lub do „prawej” podtablicy (gdy x jest większy od środkowego elementu tablicy)
- „zwyciężaj” - ustal, czy x znajduje się w wybranej podtablicy poprzez sprawdzenie elementu środkowego podtablicy, dziel i zwyciężaj właściwe podtablice na kolejne 2 podtablice aż do momentu, kiedy znajdziesz poszukiwany element lub nowo powstały „wycinek” tablicy będzie zerowej długości

Funkcja **szukajBin** (tablica **T**, liczba **x**) typu całkowitego

```
1.  // n  - rozmiar tablicy T, tablica T posortowana niemalejąco
2.  // początek, koniec, środek - indeksy
3.  początek = 0,   koniec = n-1  // krańce kolejnych podtablic
4.  while (początek <= koniec) {
5.          środek = (początek + koniec) / 2  // dzielenie całkowite
6.      if (T[środek] jest równy x)
7.      return środek
8.      else  {
9.          if (T[środek] < x)     // przejdź do „prawej” podtablicy
10.             początek = środek + 1
11.         else                 // przejdź do „lewej” podtablicy
12.             koniec = środek -1
13.     }
14. }
15. return -1
```

Należy się krótkie wyjaśnienie co do pseudokodu – w momencie, gdy T[środek] jest różny od wartości x, wybieramy jedną z dwóch podtablic, przesuwając albo *początek* **tuż za** *środek*, albo *koniec* **tuż przed** *środek* (wynika to z tego, że aktualna komórka o indeksie *środek* nie będzie już nas dalej interesowała). W ostatnim możliwym „obrocie pętli” nastąpi zrównanie indeksów: *początek* = *środek* = *koniec* – i albo element znajduje się właśnie w tej komórce T[*środek*], albo wyszukiwanie kończy się negatywnym rezultatem.

Złożoność obliczeniowa wyszukiwania binarnego jest klasy **O(lg n)**, zatem w przypadku dużych rozmiarów tablicy T czas jej przeszukiwania tą metodą jest znacznie krótszy niż metodą najprostszego wyszukiwania liniowego (gdzie złożoność jest klasy O(n)). Jest to jeden z nielicznych "poważniejszych" algorytmów, których złożoność jest aż tak niska. Przypominamy, że lg to logarytm przy podstawie 2. Zatem w przypadku ok. 10 elementów odpowiedź powinniśmy dostać po wykonaniu ok. 3 kroków (sprawdźcie "na piechote", że to prawda), zaś w przypadku ok. 1000 elementów wystarczy ok. 10 kroków (a nie 1000 - to ogromna różnica!).

Słowniki

Jedną z kluczowych grup problemów postawionych przed programistami jest wydajne poszukiwanie konkretnych informacji w zbiorach danych, które często są gromadzone w postaci ogromnych baz danych. W tym podrozdziale poznamy strukturę, dla której możliwa jest implementacja efektywnych algorytmów wyszukiwania informacji – tą strukturą jest słownik.

Mianowicie słownik (ang. *dictionary*) jest abstrakcyjnym typem danych, na którym przeprowadzane są w zasadzie trzy główne operacje:

- **dodaj** (S, x) – powoduje dodanie danej x do słownika S (zbioru danych)
- **usuń** (S, x) – powoduje usunięcie danej x ze słownika S (zbioru danych)
- **znajdź** (S,x) – wyszukuje element x w słowniku S.

Słownik (inna nazwa tego pojęcia to **tablica asocjacyjna** lub mapa) jest przykładem dynamicznego zbioru danych, to jest takiego, którego zawartość (szczególnie ilość należących do niego elementów) może zmieniać się wraz z wykonywaniem się różnych programów komputerowych. Można powiedzieć, że mapa to najprostszy i najbardziej klasyczny zbiór dynamiczny, gdyż wymaga realizacji na nim jedynie trzech wyżej wymienionych funkcji. W przypadku innych zbiorów dynamicznych niejednokrotnie konstruowane są bardziej złożone metody działające na zbiorach danych, takie jak na przykład wyszukiwanie najmniejszego/największego elementu w zbiorze.

Najczęściej zbiory danych zawierają elementy będące rekordami składającymi się z kilku, a nawet wielu pól. W tym przypadku

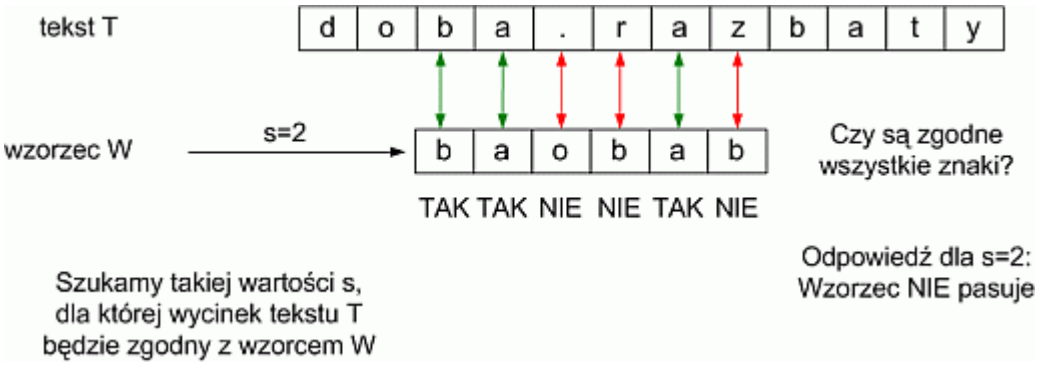
elementem wyróżniającym (identyfikującym) rekord w słowniku określa się jedno z jego pól, nazywane najczęściej mianem **klucza**.

Skoro już wiemy, że najczęściej dane typu rekordowego są elementami słowników, to najprostszym sposobem „umieszczenia" słownika w pamięci komputera jest zapisanie go w tablicy. Niestety łatwość implementacji metod dodawania, usuwania oraz wyszukiwania elementów w tablicy spotyka się z negatywnym efektem tablicowej reprezentacji mapy – mało wydajna „eksploatacja" pamięci komputera oraz stosunkowo wolne wykonywanie tych metod. Dlatego do implementacji słowników używa się częściej struktur danych, których elementami składowymi są listy – przykładem takiej struktury są na przykład **tablice haszujące** – dowiecie się o nich już w następnej lekcji. Inną bardzo efektywną strukturą danych służącą do reprezentacji słownika jest **drzewo przeszukiwań binarnych BST**, które pokażemy w lekcji 5.

PRZESZUKIWANIE TEKSTÓW

Jedną z wielu grup algorytmów, które są bardzo często wykorzystywane w codziennej praktyce, jest zbiór algorytmów do wyszukiwania wystąpień wzorca (najlepiej wszystkich) w tekście. Najczęstszym przypadkiem, kiedy potrzebne jest użycie implementacji takich algorytmów, są wszelkie programy do edycji tekstu. Spróbujmy się zatem przyjrzeć, jak przedstawia się omawiany problem.

Mamy zatem tekst T[0..N-1] oraz wzorzec W[0..M-1]. Są to ciągi znaków o liczności odpowiednio N oraz M. Znaki obu tekstów muszą należeć do wspólnego zbioru symboli. Problem wyszukiwania wzorca brzmi: należy określić, czy istnieje taki fragment tekstu T, który jest identyczny z wzorcem W. Mówiąc bardziej ściśle, chcemy wiedzieć, czy istnieje taka wartość liczbowa s, dla której T[s..M-1+s] = W[0..M-1]. Wątpliwości pomoże rozwiać przedstawiony niżej rysunek:



Natomiast zapis matematyczny faktu występowania wzorca w tekście wygląda tak:

$$\exists_{0 \le s \le N-M} \; T[s..M-1+s] == W[0..M-1], \; czyli$$
$$\forall_{0 \le i \le M-1} \; T[s+i] = W[i]$$

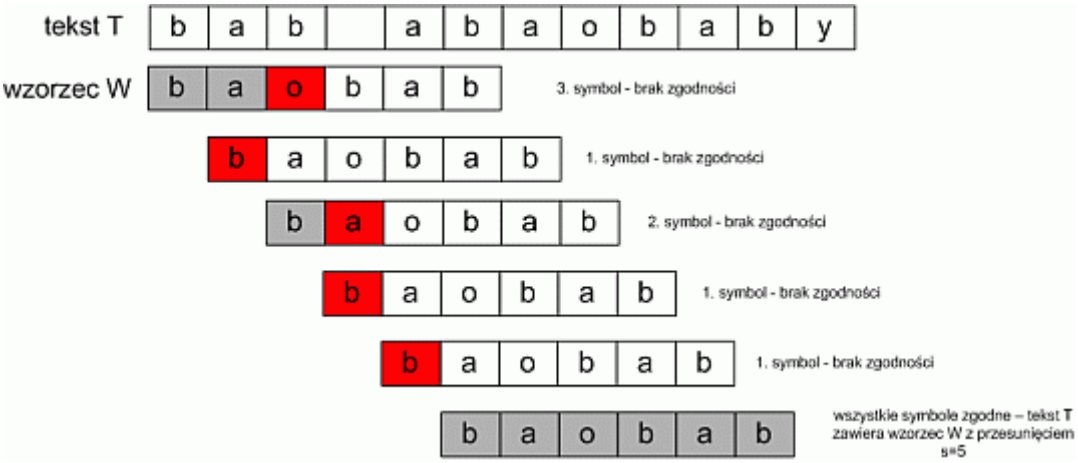
Wprowadźmy teraz przydatne pojęcia, które przydadzą się przy omawianiu poszczególnych algorytmów. Jednymi z takich pojęć jest definicja prefiksu i sufiksu. Otóż **prefiksem** słowa a będziemy nazywać takie słowo p, które będzie spełniało własność **a = pb** (gdzie p i b są słowami, których znaki występujące bezpośrednio po sobie dają łącznie słowo a). Natomiast **sufiksem** słowa a będziemy nazywać takie słowo s, które spełnia własność **a = bs** (słowa b i s łączą się podobnie jak wyżej wspomniane słowa p i b

Algorytm „naiwny" (brute-force)

Jest to najprostszy algorytm wyszukiwania wzorca. Algorytm tego typu jest bardzo prosty zarówno w zrozumieniu, jak i implementacji, jednak ma przy tym swoje wady. Otóż, jak łatwo się domyślić, jest on zdecydowanie wolniejszy niż bardziej złożone metody służące do osiągnięcia tego samego celu. Należy jednak pamiętać, że zdecydowana większość nowocześniejszych metod wyszukiwania wzorca opiera się właśnie na tym podstawowym sposobie wyszukiwania. Algorytm „naiwny" można streścić w kilku zdaniach: poczynając od przesunięcia s=0, następnie s=1,2,...N-M (gdzie N do długość tekstu T, a M to długość wzorca W), sprawdzamy, która z wartości s daje nam spełnienie warunku **T[s..M-1+s]==W[0..M-1]**. Jeśli w danej iteracji warunek nie jest spełniony, to przechodzimy do następnej wartości s.

Aby móc określić prawdziwość tego warunku, należy iteracyjnie porównać wartości T[s] i W[0], T[s+1] i W[1] ... aż do T[s+M-1] i W[M-1]. Jeśli któraś z par znaków nie będzie sobie równa, to w tym momencie algorytm stwierdza, że dane przesunięcie s jest niepoprawne i zaczynamy na nowo porównywanie symboli W i T, ale tym razem z przesunięciem o 1 większym.

Zilustrujmy algorytm poniższym przykładem:



Jak widzimy, algorytm wyszukał wzorzec w tekście dla przesunięcia s=5. W momencie niezgodności symboli na którejś z pozycji (zaznaczone na rysunku kolorem czerwonym), „okienko" wzorca jest przesuwane o 1 w kierunku końca tekstu i porównywane z odpowiednim jego wycinkiem (na rysunku będącym bezpośrednio nad nim). W najbardziej niekorzystnym układzie symboli tekstu i wzorca powyższa metoda wykona N-M+1 zewnętrznych iteracji (dla rosnącej wartości s) oraz w każdej z nich M wewnętrznych iteracji porównywania poszczególnych symboli. W takim przypadku złożoność obliczeniowa jest równa O((N-M+1)*M), czyli w przybliżeniu rzędu **O(N*M)**). W dalszej części wykładu zobaczymy, jak pod względem złożoności obliczeniowej wypadają inne algorytmy.

Pseudokod algorytmu typu brute-force jest zamieszczony poniżej:

funkcja **szukajBRF** (łańcuch **w**, łańcuch **t**) typu całkowitego

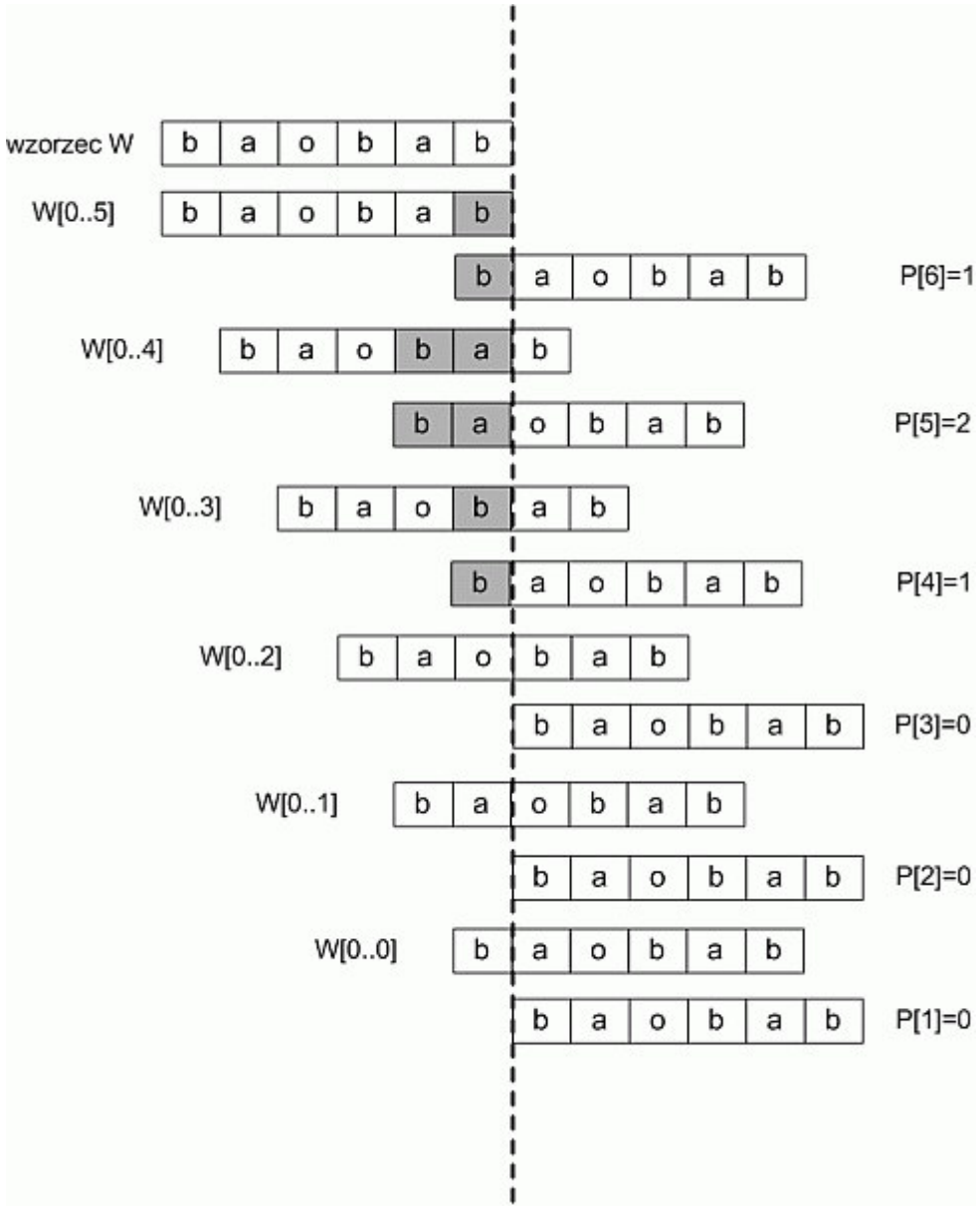
```
1.      // M, N - całkowite
2.      i=0, j=0
3.      M = długość_łańcucha()     // długość wzorca
4.      N = długość_łańcucha(t)  // długość tekstu
5.  while (j<M oraz i<N) {
6.      if (t[i] jest różne od w[j]) {
7.        i = i - a          // cofnij się do sprawdzania zgodności od pierwszej litery wzo
8.        j = -1
9.      }
10.     zwiększ i o 1
11.     zwiększ j o 1
12. }
13. if (j jest równe M)
14.     return (i-M)        // znaleziono wzorzec:
15.                        // zwracamy miejsce w łańcuchu, w którym jest pierwszy znak wzorc
16. else
17.     return -1        // nie znaleziono wzorca w tekście
```

Algorytm KMP (Knutha, Morrisa, Pratta)

Pamiętamy, że algorytm „naiwny" ma swoje wady. Podczas porównywania kolejnych symboli wzorca z symbolami tekstu nie jest wykorzystywana informacja o tej części wzorca, która już została sprawdzona. Mianowicie, gdy w danej iteracji okaże się, że symbole wzorca i tekstu nie są zgodne, algorytm przechodzi do następnej zewnętrznej iteracji zwiększając przesunięcie s o 1, a porównania symboli zaczynamy od pierwszego z nich. I w tym momencie autorzy nowego algorytmu, Knuth, Morris i Pratt postanowili nieco poprawić podstawową metodę. Otóż doszli oni do wniosku, że bylibyśmy w stanie ominąć kilka dużych iteracji (przesunięć „okna"), gdybyśmy posiadali zapamiętaną strukturę wzorca w taki sposób, dzięki któremu moglibyśmy przeskoczyć od razu o kilka przesunięć do przodu.

Do poprawnego działania algorytmu KMP potrzebna jest nam tablica przesunięć P. W takiej tablicy, która posiada tyle samo komórek co wzorzec W, element tablicy o indeksie i zawiera wartość liczbową, oznaczającą rozmiar najdłuższego prefiksu wzorca W, który jest jednocześnie właściwym sufiksem fragmentu wzorca W[0..i-1]. Wyjaśnijmy, że właściwy sufiks oznacza fragment tekstu, który nie jest jego całością. Dzięki wykonaniu wstępnych obliczeń i umieszczeniu ich w tablicy P algorytm KMP umożliwia przyspieszenie działania właściwych obliczeń i redukcję ilości wykonywanych iteracji.

Aby rozwiać wszelkie wątpliwości, na poniższym przykładzie zostanie przedstawione, jak są obliczane wartości w tablicy P.



Zatem nasza tablica P zawiera następujące wartości:

i	0	1	2	3	4	5	6
P[i]	-1	0	0	0	1	2	1

Ciekawą sytuacją jest wprowadzenie liczby –1 jako wartości P[0]. Spowodowane jest to tym, że niejednokrotnie podczas przeszukiwania tekstu zdarza się sytuacja, gdy już porównanie pierwszego znaku wzorca W ze znakiem tekstu wypada niekorzystnie. W tym momencie chcielibyśmy, by algorytm przeszedł do porównywania pierwszego znaku wzorca z kolejnym znakiem tekstu, o indeksie o 1 większym. Dlatego też algorytm KMP ustawia wartość **j = -1** (korzystając z wartości P [0]), a po przejściu do następnej iteracji oraz inkrementacji obu indeksów **i, j** kolejne porównywanie znaków zacznie się w taki sposób, jaki byśmy oczekiwali.

Z kolei element tablicy P[6] jest nam przydatny z innego powodu. Jak już wiemy, zawiera on liczbę oznaczającą długość najdłuższego prefiksu tekstu W[0..5], a więc całego wzorca, a nie tylko jego części (a dokładnie prefiksu właściwego). Informacja zawarta w tej komórce tabeli P pomaga nam kontynuować taką samą zasadę działania algorytmu KMP w momencie, gdy już znaleźliśmy wzorzec W w tekście T, ale zamierzamy szukać kolejnych (wszystkich) wystąpień wzorca w tym tekście. Dzięki temu możliwe jest pominięcie kilku zewnętrznych iteracji i "przesunięcie okna" o kilka pozycji w prawo.

Pseudokod procedury wypełniającej tablicę przesunięć możemy przedstawić następująco:

procedura **inicTablice** (łańcuch **w**, tablica **P**)

```
1.  // M - całkowite
2.  M = długość_łańcucha(w)      //długość wzorca
3.  P[0] = -1
4.  for (inicjuj i = 0 oraz j = -1; i<M; instrukcja pusta) {
5.      while (j >= 0 oraz w[i] jest różne od w[j] )
6.          j = P[j]
7.      zwiększ i o 1
8.      zwiększ j o 1
9.      P[i] = j
10. }
```

gdzie P jest właśnie zmienną tablicową (o rozmiarze równym długości wzorca) zawierającą wartości omawianej u nas tablicy przesunięć. Algorytm powyższej funkcji porównuje wzorzec z sobą samym, dla kolejnych wartości przesunięć wzorca względem siebie. Poczynając od jedynej znanej nam na starcie wartości P[0]=-1, w kolejnych krokach wyznaczane są kolejne wartości tablicy P.

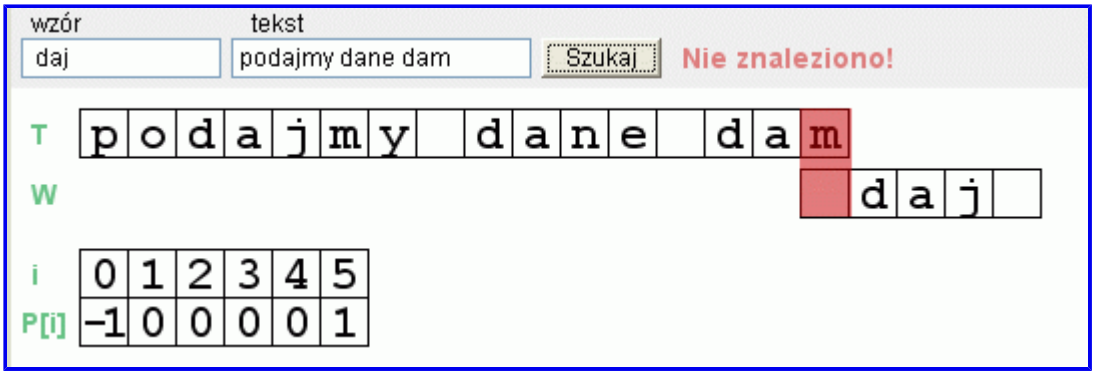
Właściwa funkcja wyszukiwania wzorca działa w sposób zbliżony do funkcji obliczającej wartości tablicy P. Startując od początku tekstu T zaczynamy go porównywać z wzorcem W, a w momencie stwierdzenie niezgodności znaków na odpowiadających sobie pozycjach, indeks j wzorca przyjmuje kolejną wartość z odpowiedniego elementu tablicy P i algorytm jest kontynuowany.

Poniższa implementacja algorytmu KMP zwraca indeks elementu w tekście T, od którego zaczyna się pierwsze znalezione wystąpienie wzorca w tekście (lub zwraca –1 w przypadku braku jakiegokolwiek wystąpienia wzorca w tekście).

funkcja **szukajKMP** (łańcuch **w**, łańcuch **t**, tablica **P**) typu całkowitego

```
1.  // M - całkowite
2.  M = długość_łańcucha(w)       // długość wzorca
3.  N = długość_łańcucha(t)      // długość tekstu
4.  inicTablice(w, P)
5.  for (inicjuj i = 0 oraz j = 0; j<M oraz i<N; zwiększ i oraz j o 1) {
6.      while (j >= 0 oraz t[i] jest różne od w[j] )
7.          j=P[j]
8.  if (j jest równe M)
9.      return (i-M)       // znaleziono wzorzec:
10.                       // zwracamy miejsce w łańcuchu, w którym jest pierwszy znak wzorca
11. else
12.     return -1       // nie znaleziono wzorca w tekście
```

W celu przybliżenia dokładnej zasady działania algorytmu KMP, przygotowaliśmy dla Państwa aplet obrazujący działanie tej metody. Aplet uruchomi się w osobnym oknie, gdy klikniecie w poniższy obrazek. Możecie sobie wpisać w jednym okienku tekst przeszukiwany, a w drugim wzorzec, i zobaczyć działanie metody KMP krok po kroku. W przykładzie jak poniżej wzorzec otoczony jest dwiema spacjami, co pozwala na wyszukiwanie odrębnych słów.



Algorytm KMP posiada inną złożoność obliczeniową niż algorytm „naiwny”. Przetwarzanie wstępne, czyli wyznaczenie tablicy przejść P wymaga czasu O(M), natomiast właściwy algorytm wyszukiwania wystąpień wzorca w tekście posiada złożoność O(N). Jak zatem łatwo zauważyć, sumaryczny czas wszystkich obliczeń niezbędnych do znalezienia wzorca metodą KMP jest rzędu **O(M+N)**, co jest wynikiem lepszym, niż złożoność obliczeniowa O(N*M), jaką posiada algorytm typu *brute-force*. Złożoność O(N) właściwej części algorytmu KMP wynika z faktu, że w przeciwieństwie do algorytmu "naiwnego", wewnętrzne pętle porównywania znaków niekoniecznie rozpoczynają iteracje od j=0 (a wtedy jest maksymalnie m porównań w każdej z zewnętrznych pętli), lecz wykorzystują wartości z tablicy P, które są przypisywane zmiennej iteracyjnej j, co ogranicza ilość wykonanych zbędnych obliczeń i porównań znaków.