Metoda programowania „dziel i zwyciężaj” – część I

Metoda „dziel i zwyciężaj” (ang. *divide and conquer*) polega na podzieleniu problemu na mniejsze (podproblemy), a następnie rozwiązaniu ich, co ostatecznie prowadzi do rozwiązania problemu pierwotnego. Zatem składa się ona z trzech etapów:

1. Podział problemów na podproblemy.
2. Rozwiązanie każdego z podproblemów.
3. Scalenie rozwiązanych podproblemów w całość (ogólne rozwiązanie problemu).

Prześledźmy metodę na klasycznym przykładzie przeszukiwania binarnego uporządkowanego ciągu liczbowego. Idea algorytmu opiera się na wyznaczeniu wyrazu środkowego i porównaniu go z szukaną liczbą. Jeśli obie wartości są równe, to szukany wyraz został odnaleziony, a jeśli nie, to na podstawie porównania poszukujemy tego wyrazu albo wśród pierwszej połowy ciągu (gdy poszukiwany wyraz jest mniejszy od środkowego), albo przeciwnie. Postępujemy tak do czasu znalezienia poszukiwanej liczby bądź ustalenia, że nie było jej w ciągu.

Gdybyśmy przeszukiwali ciąg liczbowy liniowo (wyraz po wyrazie), wówczas złożoność obliczeniowa algorytmu wyniosłaby $O(n)$, gdzie n oznacza liczbę wyrazów ciągu. Ponieważ skorzystaliśmy z metody „dzieli i zwyciężaj”, to jego złożoność wyniesie jedynie $O(\log_2 n)$, co oznacza bardzo duże przyspieszenie obliczeń oraz zmniejszenie użycia zasobów komputera.

Jeśli mamy do dyspozycji nieuporządkowany ciąg liczbowy i mamy w nim jednocześnie znaleźć element minimalny i maksymalny, możemy niezależnie wyszukać oba elementy, porównując ze sobą kolejne liczby. Złożoność obliczeniowa każdego przebiegu algorytmu wyniesie $(n-1)$, zatem sumaryczna złożoność wyniesie $2(n-1)$. Czy da się znaleźć i zastosować algorytm o mniejszej złożoności obliczeniowej? Z pomocą przychodzi metoda „dziel i zwyciężaj”.

Weźmy pod uwagę ciąg liczbowy $(2, 3, 4, 3, 6, 7, 1, 0, 3, 3, 6, 9, 2)$, który składa się z 13 elementów. W pierwszym etapie porównajmy kolejne liczby, ale parami, i utwórzmy dwa podciągi. Ponieważ mamy nieparzystą liczbę elementów, ostatni z nich, czyli liczbę 2, zaliczymy do obu podciągów, zatem oba podciągi będą miały po 7 elementów. W jednym z nich na pewno będzie kandydat na minimum, a w drugim na maksimum, co obrazuje poniższa tabelka.

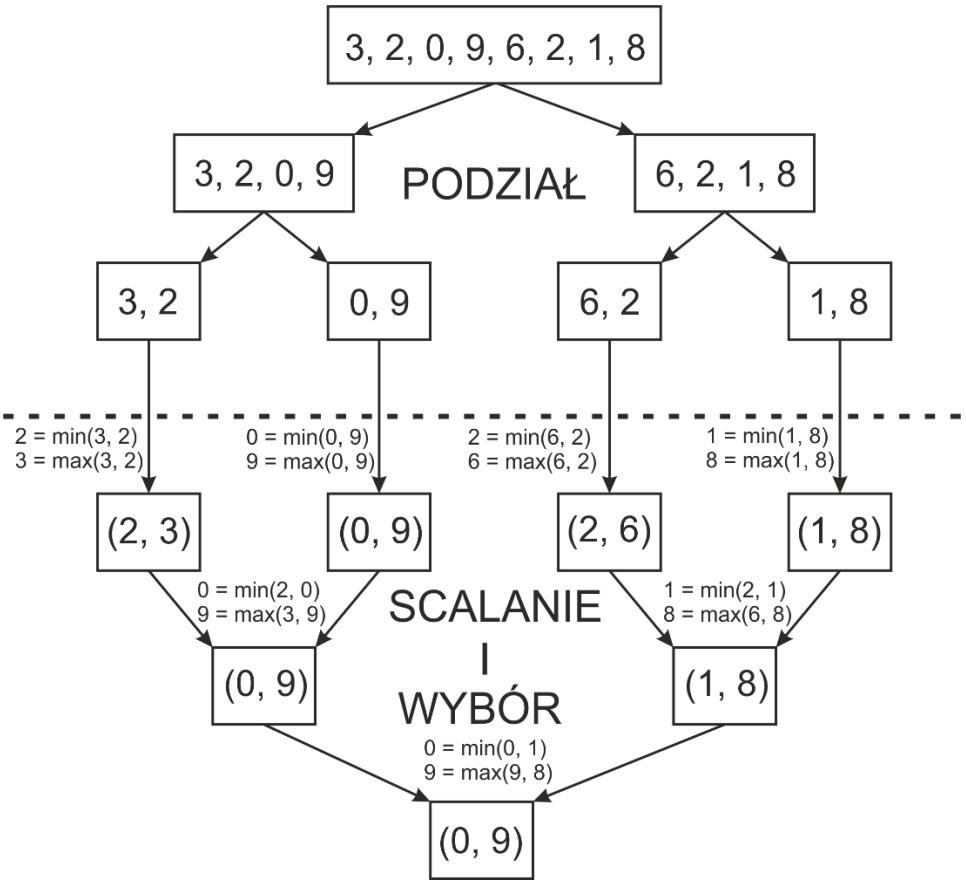
MIN	2	3	6	0	3	6	2
Porównywane pary liczb	$\mathbf{2 \leq 3}$	$\mathbf{4 > 3}$	$\mathbf{6 \leq 7}$	$\mathbf{1 > 0}$	$\mathbf{3 \leq 3}$	$\mathbf{6 \leq 9}$	**2**
MAX	3	4	7	1	3	9	2

Następnie możemy użyć klasycznego algorytmu do przejrzenia każdego podciągu z osobna. Dzięki temu łącznie uzyskamy złożoność obliczeniową rzędu ok. $\frac{3}{2}n$ (precyzyjnie zależną od tego, czy ciąg pierwotny miał parzystą bądź nieparzystą liczbę elementów), przy czym da się w sposób ścisły udowodnić, że otrzymany algorytm jest optymalny (najlepszy z możliwych w danych warunkach). W języku C++ funkcja realizująca ten algorytm w sposób iteracyjny może mieć postać (brak komentarzy jest zamierzony):

```
void oblicz(double T[], int n, double &min, double &max)
{
 int dl, i;
 if (n % 2) dl = n - 2; else dl = n - 1;
 if (T[0] <= T[1])
 {
  min = T[0];
  max = T[1];
 }
 else
 {
  min = T[1];
  max = T[0];
 }
 i = 2;
```

```
 while (i < dl)
 {
  if (T[i] <= T[i + 1])
  {
   if (T[i] < min) min = T[i];
   if (T[i + 1] > max) max = T[i + 1];
  }
  else
  {
   if (T[i + 1] < min) min = T[i + 1];
   if (T[i] > max) max = T[i];
  }
  i += 2;
 }
 if (n % 2)
 {
  if (T[n - 1] < min) min = T[n - 1];
  if (T[n - 1] > max) max = T[n - 1];
 }
}
```

W oparciu o powyższą analizę da się również zbudować algorytm rekurencyjny. Tym razem weźmy pod uwagę parzysty, 8-elementowy ciąg liczb $(3, 2, 0, 9, 6, 2, 1, 8)$. Algorytm, który wyznaczy minimalną i maksymalną liczbę tego ciągu będzie miał przebieg pokazany na rysunku.



Z3.

W języku C# (Java, Python lub C++) napisz program, który zrealizuje algorytm poszukujący minimalny i maksymalny wyraz ciągu liczbowego w sposób rekurencyjny (przedstawiony na rysunku powyżej). Przeprowadź analizę złożoności obliczeniowej zastosowanego algorytmu. **Termin: 2023-03-24 (do północy).**

Metoda „dziel i zwyciężaj” znajduje także szerokie zastosowanie w porządkowaniu zbiorów. Opiera się na niej np. sortowanie przez scalanie oraz sortowanie szybkie (poniżej krótkie przypomnienie wiadomości na temat sortowania).

Sortowanie (ang. *sorting*) – uporządkowanie danych lub informacji (kart bibliotecznych, słów w encyklopedii, dat urodzin, numerów telefonów itp.) poprzez ustawienie ich w określonej kolejności (najczęściej rosnąco lub malejąco).

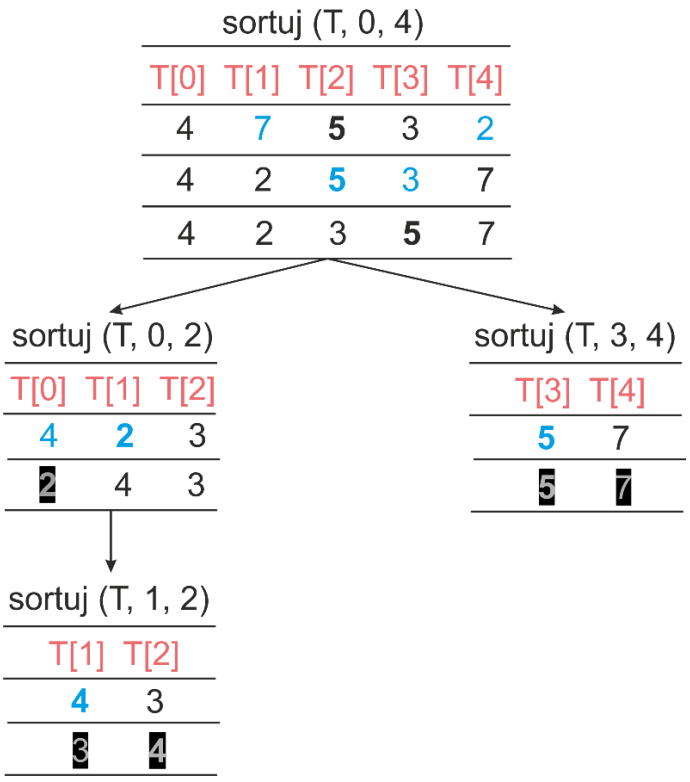
Dzięki posortowaniu operacje na uporządkowanym zbiorze stają się łatwiejsze i szybsze. Dotyczy to m.in.:

- sprawdzenia, czy element o ustalonej wartości cechy znajduje się w zbiorze,
- znalezienia elementu w zbiorze, jeśli w nim jest,
- dołączenia nowego elementu w odpowiednie miejsce, aby zbiór pozostał nadal uporządkowany.

Istnieje wiele metod sortowania, wśród nich najpopularniejsze to:

1. Bąbelkowe (ang. *bubble sort*) – stabilne
2. Przez wstawianie (ang. *insertion sort*) – stabilne
3. Przez scalanie (ang. *merge sort*) – stabilne
4. Przez wybór (ang. *selection sort*) – niestabilne
5. Szybkie (ang. *quick sort*) – niestabilne
6. Stogowe (*heap sort*) – niestabilne

Przyjrzyjmy się zatem rysunkowi obrazującemu sortowanie szybkie, które ma uporządkować rosnąco ciąg liczb $(4, 7, 5, 3, 2)$ w sposób rekurencyjny. Kolorem niebieskim zaznaczono liczby przeznaczone do zamiany, pogrubieniem wyraz środkowy, zaś elementy już posortowane wyróżniono czarnym tłem.



W kolejnym etapie posortowane podciągi należałoby scalić, podobnie jak przy jednoczesnym wyszukiwaniu najmniejszego i największego elementu zbioru.

Specyfikacja i lista kroków algorytmu sortowania szybkiego

Wejście:

n – liczba naturalna większa od zera (liczba elementów tablicy T)

$T[0 \ldots n-1]$ – n-elementowa tablica (jednowymiarowa) zawierająca nieposortowany ciąg liczb rzeczywistych

Wyjście:

$T[0 \ldots n-1]$ – n-elementowa tablica (jednowymiarowa) zawierająca posortowany ciąg liczb rzeczywistych

Lista kroków:

K0. Wczytaj n, $T[0 \ldots n-1]$.

K1. Jeśli $lewy < prawy$, przypisz $srodek = T[(lewy + prawy)/2]$. W przeciwnym wypadku zakończ algorytm.

K2. Przegrupuj wyrazy ciągu $T[lewy, prawy]$ w dwa podciągi wykorzystując wyraz środkowy $srodek$. Po podziale wyraz $srodek$ znajdzie się na pozycji elementu $T[k]$, dla k spełniającego warunek $lewy \leq k \leq prawy$. Elementy zawarte w podciągu $T[lewy, k]$ będą od niego nie większe, a w podciągu $T[k+1, prawy]$ będą od niego nie mniejsze.

K3. Uruchom ten algorytm z parametrami $(T, lewy, k)$.

K4. Uruchom ten algorytm z parametrami $(T, k+1, prawy)$.

Z4. (dla ambitnych)
W języku C# (Java, Python lub C++) napisz program, który porównuje co najmniej trzy z sześciu wymienionych metod sortowania (im więcej metod, tym więcej przyznanych punktów) pod kątem złożoności obliczeniowej. Program powinien porównywać czas wykonania wszystkich algorytmów dla tych samych danych wejściowych oraz podać liczbę porównań i zamian dla każdego z nich, przy czym operacją dominującą (zajmującą więcej czasu procesorowi) jest tutaj zamiana. **Termin: 2023-03-24 (do północy).**