MINUT 2021 (3), s. 153–165
ISSN 2719-3063

Robert TUTAJEWICZ[1]

[1]Katedra Informatyki Stosowanej, Politechnika Śląska, ul. Akademicka 16, 44-100 Gliwice

Ocena efektywności algorytmu na przykładzie szukania indeksu minimum i maksimum

Streszczenie. W artykule na przykładzie problemu poszukiwania indeksów największego i najmniejszego elementu w tablicy pokazano, w jaki sposób należy analizować algorytmy pod kątem oceny ich efektywności czasowej. Przedstawiono także, czym jest złożoność obliczeniowa algorytmu i jak ją szacować uwzględniając wpływ operacji dominującej na czas wykonania. Zaproponowano 6 różnych wersji algorytmów rozwiązujących postawione zadanie i dla każdej z nich policzono złożoność danego algorytmu.
Słowa kluczowe: efektywność algorytmów, optymalizacja programu, poszukiwanie minimum i maksimum, złożoność obliczeniowa, operacja dominująca.

1. Wstęp

Każdy programista, pisząc program, stara się go napisać możliwie jak najlepiej. Jak jednak ocenić, który program jest lepszy? Co przyjąć za miernik jakości programu? Najczęściej jako kryterium takiej oceny wskazuje się czas trwania obliczeń. Czy jednak w sytuacji, gdy dwóch programistów chwali się swoimi programami i pierwszy mówi, że jego program potrzebuje na obliczenie pewnego zadania 15 sekund a drugi odpowiada, że jego rozwiązanie wyznacza wynik tego zadania w 10 sekund, możemy być pewni, że drugi napisał lepszy program? Otóż nie. Nie wiemy, czy oba programy były wykonywane na takim samym komputerze, być może ten drugi został uruchomiony na znacznie szybszej jednostce. Także nic nie wiemy o systemie operacyjnym, pod kontrolą którego wykonywano te programy, ani o językach programowania i kompilatorach, jakich użyto do skompilowania programów. Nawet gdyby system operacyjny był ten sam, to być może priorytety obydwu wykonywanych programów były różne. A gdyby użyto tego samego kompilatora, to kompilacja mogła odbywać się przy włączonych różnych opcjach optymalizacji. Ale nawet jeśliby wszystkie wymienione czynniki były identyczne a programy były wykonywane na tym samym komputerze, to warunki ich wykonania nie byłyby zapewne identyczne (np. w różnych chwilach wykonania pojawiałyby się różne przerwania). Tym samym pomiar czasu wykonania nie wystarcza, by z całą pewnością powiedzieć, że dane rozwiązanie jest lepsze od innego.

W praktyce nie porównuje się samych programów a algorytmy, jakie w tych programach zostały użyte. Dzięki temu wszystkie wymienione wyżej czynniki zaburzające wynik porównania nie mają wpływu na

Autor korespondencyjny: R. Tutajewicz (robert.tutajewicz@polsl.pl).
Data wpłynięcia: 15.09.2021.

rezultat, który uzyskamy. Ale jak ocenić, który algorytm jest szybszy? Mówi o tym cecha algorytmu nazywana efektywnością czasową. Określa ona, na ile wydajny jest dany algorytm. Miarą efektywności algorytmów jest złożoność obliczeniowa. Zatem, aby porównać algorytmy, należy określić ich złożoność obliczeniową. W niniejszym artykule na przykładzie dosyć prostego problemu zostanie pokazane, jak to zrobić.

Rozpatrywanym tu problemem jest wyznaczenie indeksu najmniejszego i największego elementu w pewnym skończonym n-elementowym ciągu liczb rzeczywistych a_n. Zostanie przedstawionych kilka algorytmów realizujących to zadanie. Dla każdego z nich zostanie także wyznaczona złożoność obliczeniowa. Dzięki temu będzie możliwe porównanie ze sobą poszczególnych algorytmów.

2. Najprostsze rozwiązanie

Jak już wspomniano, zadanie polega na wyznaczeniu indeksu najmniejszego i największego elementu w pewnym skończonym zbiorze liczb rzeczywistych a_n, gdzie $n \geqslant 1$. Zbiór ten będzie reprezentowany za pomocą tablicy liczb rzeczywistych `a[n]`. Przykładowe deklaracje stałej `n` i tablicy `a[]` jako zmiennej globalnej w języku C przedstawiono na listingu 1. Przyjęto tam, że `n` jest równe 10 ale nic nie stoi na przeszkodzie, by tablica liczyła znacznie większą liczbę elementów.

```
1    const int n = 10;
2    const double a[n] = { 4.1, 6.2, -12.3, 3.0, -8.63, 11, 7.4, -5.8, 9.01, 10.4 };
```

Listing 1. Deklaracje stałej n i zmiennej a

Należy zauważyć, że mimo, iż odpowiednie algorytmy zostaną zaprezentowane przy założeniu, że elementami zbioru są liczby rzeczywiste, algorytmy te można stosować dla innych rodzajów danych. W rzeczywistości wystarcza, aby w zbiorze istniała relacja częściowego porządku. W przypadku, gdy najmniejszy bądź największy element powtarza się kilkakrotnie, jako wynik wybierany jest element o najmniejszej wartości indeksu (bierze się pod uwagę pierwsze wystąpienie danego elementu w ciągu).

Dana jest zatem n-elementowa tablica liczb rzeczywistych indeksowana od 0, a zadaniem algorytmu jest znalezienie indeksów elementów minimalnego i maksymalnego spośród wszystkich liczb tworzących tablicę. Jak to zrobić? Jedną z metod poszukiwania algorytmów jest podział zadania na mniejsze i szukanie algorytmów rozwiązujących te mniejsze problemy z osobna. W tym konkretnym zadaniu należy znaleźć miejsca w tablicy, pod którymi znajdują się dwie liczby – najmniejsza i największa. Można zatem podzielić całe zadanie na dwa podzadania: znalezienie indeksu minimum i znalezienie indeksu maksimum. Najpierw zostanie omówione rozwiązanie pierwszego z podzadań.

Jeżeli dany jest zbiór jednoelementowy, to jedyny element tworzący ów zbiór jest jednocześnie minimum w tym zbiorze, zatem odpowiedzią jest indeks tego elementu równy 0. Korzystając z metody indukcji matematycznej możemy wyprowadzić algorytm poszukiwania indeksu minimum w zbiorze liczącym więcej niż 1 element (zbiorze n-elementowym). Załóżmy, że znamy położenie (indeks) najmniejszego elementu w zbiorze złożonym z $k-1$ początkowych elementów zbioru n-elementowego (gdzie $k \leqslant n$; oznaczmy ten indeks przez `i_min`). Czy na tej podstawie można znaleźć indeks minimum w zbiorze k-elementowym powstałym z poprzedniego poprzez dodanie do niego jednego elementu (nazwijmy go elementem k-tym, czy zgodnie z konwencją przyjętą w języku C, elementem `a[k-1]`). Oczywiście, że tak. Istnieją tylko dwie możliwości. Albo ostatnio dodany element będzie najmniejszy w zbiorze i wtedy będzie stanowił minimum w całym zbiorze (i tym samym jego indeks będzie indeksem najmniejszego elementu) albo dodany element będzie większy (lub równy, o ile dopuszczamy powtórzenia) od przynajmniej jednego elementu ze

zbioru liczącego pierwsze $k-1$ elementów (precyzyjniej elementu o indeksie `i_min`). W tym drugim przypadku najmniejszym elementem zbioru pozostanie element o indeksie `i_min`. Rozumowanie to prowadzi do wniosku, że opisany algorytm postępowania pozwoli zawsze znaleźć indeks elementu najmniejszego w zbiorze, tym samym została udowodniona poprawność tego algorytmu.

Podobnie należałoby postąpić podczas szukania indeksu elementu największego. Po połączeniu rozwiązań obydwu podzadań otrzymuje się algorytm zapisany w języku programowania C a przedstawiony na listingu 2. Autor zakłada, że Czytelnik zna co najmniej podstawy tego języka. Jeśli tak nie jest, autor sugeruje skorzystanie z jednego z wielu dostępnych w Internecie kursów (np. [4] czy [5]).

```
1  void minmax1( double a[], int n, int &i_min, int& i_max)
2  {
3      // poszukiwanie indeksu elementu najmniejszego
4      i_min = 0;
5      for (int i = 1; i < n; i++) {
6          if (a[i] < a[i_min])
7              i_min = i;
8      }
9      // poszukiwanie indeksu elementu największego
10     i_max = 0;
11     for (int i = 1; i < n; i++) {
12         if (a[i] > a[i_max])
13             i_max = i;
14     }
15 }
```

Listing 2. Szukanie indeksu minimum i maksimum rozbite na dwie pętle

Algorytm został zapisany w postaci funkcji `minmax1`. Pierwsza linia kodu przedstawionego na listingu 2 tworzy nagłówek funkcji. Słowo `void` oznacza, że funkcja nie zwraca wyniku, a jedynie wykonuje działania opisane kodem między nawiasami klamrowymi. Po słowie `void` następuje nazwa funkcji oraz w nawiasach okrągłych umieszczona jest informacja o jej parametrach. Do funkcji `minmax1` przekazywane są cztery parametry. Pierwszy, nazwany `a[]`, reprezentuje tablicę, której zawartość jest przeglądana w poszukiwaniu indeksów najmniejszego i największego elementu. Rola parametru `n` jest dość oczywista, to liczba elementów tablicy `a[]`. Parametry `i_min` i `i_max` to indeksy odpowiednio najmniejszego i największego elementu tablicy `a[]`. Dwa pierwsze parametry przekazywane są przez wartość, pozostałe dwa przez referencję (stąd symbol `&` przed nazwą odpowiedniego parametru). Przekazywanie parametrów przez referencję powoduje, że efekty zmian wykonanych wewnątrz funkcji widoczne są po jej zakończeniu na zewnątrz. Ponieważ funkcja `minmax1` ma za zadanie odnaleźć indeksy najmniejszego i największego elementu, a także znalezione wartości indeksów powinny być dostępne po zakończeniu wykonywania funkcji, należy użyć właśnie tej formy przekazywania parametrów (przekazywania przez referencję). Więcej informacji o przekazywaniu parametrów w C można znaleźć w [6], zaś [8] opisuje, jak przekazywane są do funkcji tablice.

Działanie funkcji zostało podzielone na dwie części. Najpierw (linie 4-8) przeszukiwana jest tablica `a[]` w celu znalezienia indeksu elementu najmniejszego `i_min`, a później (linie 10-14) poszukiwany jest indeks elementu największego `i_max`. Warto dodać, że w przypadku tablicy jednoelementowej (gdy $n = 1$) wnętrza pętli `for` (linie 5 i 11) nie są w ogóle wykonywane, gdyż od razu nie jest spełniony warunek kontynuacji `i < n`. Dzięki temu funkcja `minmax1` zwraca w tym przypadku prawidłowy wynik (`i_min = i_max = 0`).

Aby ocenić efektywność czasową przedstawionego na listingu 2 algorytmu należy określić, jakie i ile operacji będzie trzeba wykonać, by znaleźć indeks największego i najmniejszego elementu w n-elementowej tablicy `a[]`. Już nawet w przypadku tak prostego algorytmu widać, że tworzy go wiele różnych operacji.

Mamy tu do czynienia z operacjami nadania wartości początkowej zmiennej (linie 4 i 10 oraz inicjalizacja zmiennej `i` sterującej wykonaniem pętli `for` w liniach 5 i 11), przypisania zmiennej wartości innej zmiennej (linie 7 i 13), inkrementacji zmiennych sterujących w pętli (linie 5 i 11) i operacjami porównania zmiennych (w ramach instrukcji warunkowych `if` w liniach 6 i 12 oraz przy sprawdzaniu warunków zakończenia wykonywania pętli `for` w liniach 5 i 11). Co więcej to, ile razy będzie wykonana dana operacja, nie jest stałe i zależy od liczby przetwarzanych elementów n, ale także od kolejności, w jakiej będą rozmieszczone wartości w tablicy `a[]`. Wykonanie poszczególnych operacji wymaga wreszcie różnego czasu. Zatem uwzględnienie wszystkich operacji we wzorze wyrażającym zależność czasu wykonania algorytmu od podanych wcześniej czynników spowoduje, że ów wzór będzie bardzo skomplikowany i na niewiele się zda przy określaniu rzeczywistej efektywności algorytmu.

Zwykle wybiera się spośród wszystkich operacji tworzących algorytm jedną operację (lub ewentualnie rzadziej kilka operacji), która w największym stopniu wpływa na czas wykonania algorytmu i liczy się tylko liczbę wykonań tej wybranej operacji, nazywanej zwykle operacją dominującą. Operacja dominująca to operacja charakterystyczna dla danego algorytmu, najczęściej zajmująca w jego wykonaniu najwięcej czasu. Wybór operacji dominującej zależy od problemu, jaki rozwiązuje dany algorytm oraz od metody rozwiązania. W algorytmach mnożenia macierzy operacjami dominującymi są zwykle operacje dodawania i mnożenia. W przypadku algorytmów sortowania zwykle operacją dominującą jest operacja porównania, ale na przykład w przypadku algorytmu sortowania przez zliczanie, gdzie porównanie praktycznie nie występuje, za operację dominującą przyjmuje się dostęp (odczyt lub modyfikację wartości) do elementów sortowanej tablicy.

W zadaniu wyszukiwania indeksu najmniejszego i największego elementu jako operację dominującą przyjęto operację porównania elementów tablicy. Warto zauważyć, że oprócz porównania elementów tablicy, występujących w liniach 6 i 12 algorytmu przedstawionego na listingu 2, w algorytmie tym występują także inne porównania (w liniach 5 i 11), związane z organizacją działania pętli `for`. Operacje te dotyczą porównania wartości liczb całkowitych a nie elementów tablicy i z tego powodu nie będą uwzględniane. Zwykle porównanie liczb całkowitych realizowane jest przez procesor znacznie szybciej niż porównanie liczb rzeczywistych, a takie występują w tablicy `a[]`. Ponadto warto pamiętać, że przytoczony algorytm pozwala porównywać zgromadzone w tablicy elementy dowolnego typu, a nie tylko liczby rzeczywiste. Gdyby na przykład w tablicy umieszczono teksty lub liczby zespolone, wystarczyłoby tylko nieznacznie zmodyfikować ten algorytm wprowadzając operację porównania elementów odpowiedniego typu, lecz sama postać i idea algorytmu nie uległaby znaczącej zmianie. W dalszej części niniejszego opracowania za operację dominującą została przyjęta operacja porównania elementów tablicy.

Operacja porównania jest stosowana w algorytmie dwukrotnie, pierwszy raz w linii 6 i drugi w linii 12. W obydwu przypadkach porównanie wykonywane jest wewnątrz pętli `for` począwszy od wartości zmiennej sterującej `i` równej 1, aż do `n-1`. Co daje n - 1 wykonań porównania tak w pierwszej, jak i drugiej pętli. Można zatem podać wzór na łączną liczbę wykonań operacji dominującej w omawianym algorytmie

$$L_1(n) = 2(n-1). \tag{1}$$

Ze wzoru wynika, że liczba wykonań operacji dominującej liniowo zależy od rozmiaru przetwarzanej tablicy n. Taka zależność wydaje się dość oczywista, im większa tablica, tym dłużej będziemy czekać na znalezienie w niej indeksów najmniejszego i największego elementu. Algorytm przedstawiony na listingu 2 można jednak łatwo ulepszyć. Naturalnym i prostym udoskonaleniem algorytmu jest umieszczenie obydwu porównań wewnątrz jednej pętli. W efekcie uzyskuje się nieco zmodyfikowaną wersję algorytmu, przedstawioną na listingu 3.

```
1  void minmax2( double a[], int n, int &i_min, int& i_max)
2  {
3      // poszukiwanie indeksu elementu najmniejszego i największego w jednej pętli
4      i_min = i_max = 0;
5      for (int i = 1; i < n; i++) {
6          if (a[i] < a[i_min])
7              i_min = i;
8          if (a[i] > a[i_max])
9              i_max = i;
10     }
11 }
```

Listing 3. Szukanie indeksu minimum i maksimum w jednej pętli

Wprowadzona modyfikacja w ogóle nie zmienia liczby wykonań operacji dominującej. O ile w poprzedniej wersji były dwie pętle i w każdej z nich operacja dominująca była wykonywana $n-1$ razy, tak teraz jest jedna pętla wykonywana $n-1$ razy, ale w każdym jej przebiegu wykonywane są dwie operacje dominujące, czyli operacje porównania elementów tablicy (linie 6 i 8).

Mimo, że z punktu widzenia efektywności obydwa algorytmy mają tę samą liczbę operacji dominujących, w praktyce algorytm przedstawiony na listingu 3 wykonana się zwykle minimalnie szybciej. Jest to związane z mniejszą liczbą wykonań operacji sterujących pętlą `for` i przede wszystkim sposobem organizacji dostępu do pamięci we współczesnych komputerach (wykorzystanie mechanizmu pamięci podręcznej). Są to jednak czynniki o charakterze ściśle technicznym, związane z realizacją programów, a nie samymi algorytmami, dlatego też nie poświęcono im tu więcej uwagi.

3. Pierwsze udoskonalenie

Aby poprawić efektywność algorytmu należy go tak zmodyfikować, by zmniejszyć liczbę wykonywanych operacji dominujących, czyli w tym przypadku operacji porównania elementów tablicy `a[]`. W rozważanym algorytmie każdy kolejny element tablicy jest najpierw porównywany z dotychczas najmniejszym, a potem z dotychczas największym (linie 6 i 8 na listingu 3). Tymczasem jeśli ów element okaże się być mniejszy od dotychczas najmniejszego, to na pewno nie będzie jednocześnie większym od dotychczas największego. A zatem wykonywanie drugiego porównania ma sens tylko wtedy, gdy pierwsze zakończyło się negatywnie. Spostrzeżenie to prowadzi do skonstruowania kolejnej wersji algorytmu poszukującego indeksów najmniejszego i największego elementu w tablicy. Wersję tę prezentuje listing 4.

```
1  void minmax3( double a[], int n, int &i_min, int& i_max)
2  {
3      // poszukiwanie indeksu elementu najmniejszego i największego w jednej pętli
4      i_min = i_max = 0;
5      for (int i = 1; i < n; i++) {
6          if (a[i] < a[i_min])
7              i_min = i;
8          else if (a[i] > a[i_max])
9              i_max = i;
10     }
11 }
```

Listing 4. Algorytm ze zmniejszoną liczbą porównań

Jedyną różnicą w tym kodzie w stosunku do wersji z listingu 3 jest poprzedzenie instrukcji `if` w linii 8 słowem kluczowym `else`. Powoduje to, że drugie porównanie jest wykonywane tylko wtedy, gdy pierwsze

zakończyło się niepowodzeniem. Tym samym liczba porównań niewątpliwie zmniejszyła się, pozostaje tylko pytanie, o ile.

W przypadku poprzednio rozważanych wersji algorytmu, liczba wykonań operacji porównania elementów tablicy zależała wyłącznie od rozmiaru tejże tablicy. W tym przypadku istotnego znaczenia nabiera także kolejność, w jakiej umieszczono wartości w tablicy. Jeżeli dane będą uporządkowane malejąco (i bez powtórzeń), porównanie wykonywane w linii 6 zawsze zakończy się powodzeniem (gdyż każdy kolejny element będzie mniejszy od wszystkich poprzednich) i w konsekwencji porównanie z linii 8 nigdy nie będzie wykonywane. W takim przypadku łącznie zostanie zatem wykonane zaledwie $n-1$ porównań, co daje wynik dwukrotnie lepszy niż w poprzednio prezentowanych algorytmach. Jednakże, jeśli pierwszy element w tablicy `a[]` będzie najmniejszym, drugie porównanie zawsze będzie wykonywane, czyli liczba porównań znów wyniesie $2(n-1)$.

Aby ocenić, o ile najnowsza wersja algorytmu jest lepsza od poprzednich, należy policzyć, ile średnio porównań zostanie wykonanych. Wzór, będący odpowiedzią na tak postawione pytanie, najłatwiej uzyskać odejmując od maksymalnej liczby wykonywanych porównań średnią liczbę tych, które nie zostaną wykonane. Porównanie w linii 8 nie jest wykonywane wtedy, gdy spełniony jest warunek sprawdzany w linii 6. Do porównania z linii 8 nie dojdzie zatem wtedy, gdy kolejny rozważany element jest mniejszy od wszystkich go poprzedzających. Jako, że przetwarzanie w pętli rozpoczyna się od drugiego elementu tablicy (elementu `a[1]`), jeżeli założymy losowe rozmieszczenie elementów, drugi element będzie mniejszy od pierwszego z prawdopodobieństwem $\frac{1}{2}$. Aby nie doszło do drugiego porównania, trzeci element `a[2]` musi być mniejszy od obydwu poprzednich, a to ma miejsce z prawdopodobieństwem $\frac{1}{3}$. Z kolei z prawdopodobieństwem $\frac{1}{4}$ element czwarty będzie mniejszy od wszystkich go poprzedzających i tak dalej. Dla ostatniego elementu w tablicy (elementu `a[n - 1]`), prawdopodobieństwo, że nie dojdzie do wykonania drugiego porównania wynosi $\frac{1}{n}$. Tym samym średnia liczba pominiętych porównań to

$$\frac{1}{2}+\frac{1}{3}+\frac{1}{4}...+\frac{1}{n}=\sum_{i=2}^{n}\frac{1}{i}. \tag{2}$$

Uzyskany wynik przypomina ciąg liczb harmonicznych H_n, z tą różnicą, że w tym przypadku zaczynamy od drugiego wyrazu. W 1734 roku Leonhard Euler wykazał [10], że

$$\lim_{n\to\infty}(H_n-\ln n)=\gamma, \tag{3}$$

gdzie γ to tzw. stała Eulera, równa w przybliżeniu 0,5772.

Poszukiwaną sumę można zatem oszacować (dla dużych wartości n) jako

$$\sum_{i=2}^{n}\frac{1}{i}=\ln n+\gamma-1. \tag{4}$$

Oznacza to, że średnia liczba wykonanych operacji porównania wynosi

$$2(n-1)-\sum_{i=2}^{n}\frac{1}{i}\approx 2n-\ln n-\gamma-1. \tag{5}$$

Ze wzrostem n funkcja logarytm naturalny rośnie znacznie wolniej niż funkcja liniowa, zatem wpływ wprowadzonej poprawki na czas realizacji algorytmu będzie tym mniej znaczący, im większy jest rozmiar tablicy. Gdy wzór określający liczbę operacji dominujących w funkcji n tworzy kilka składników, pod uwagę bierze się wyłącznie składnik najbardziej znaczący, gdyż przy dużych wartościach n to ten składnik

ma największy wpływ na łączny czas przetwarzania [2] [3]. Po pominięciu mniej znaczących składników we wzorze 5, można przyjąć więc, że liczba operacji dominujących jest rzędu $2n$. Zatem także tym razem efekt wprowadzonej poprawki okazał się mizerny.

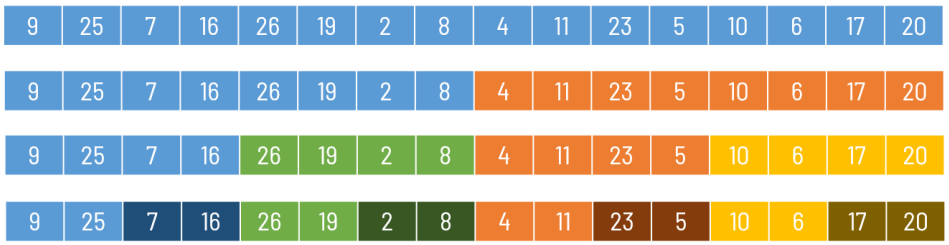
Warto zauważyć, że do prawidłowego wykonania postawionego zadania konieczny jest przynajmniej jeden dostęp do każdego elementu tablicy i porównanie go z innym, nic bowiem nie wiadomo o rozmieszczeniu elementów w tablicy `a[]` i całkowicie pomijając którykolwiek z nich, można by pominąć poszukiwany element najmniejszy bądź największy. Zatem jakikolwiek zaproponowany algorytm musi cechować się złożonością liniową. Jedyne co można zrobić, to obniżyć wartość współczynnika występującego przy n.

4. Metoda dziel i zwyciężaj

Jednym z często stosowanych podejść przy poszukiwaniu wydajnych algorytmów jest zastosowanie metody dziel i zwyciężaj. Jej istota sprowadza się do dzielenia zbioru danych na mniejsze fragmenty, rozwiązaniu zadania dla mniejszych fragmentów i wykorzystaniu wyników tych rozwiązań do znalezienia rozwiązania dla całego, dużego zbioru danych. Oczywiście do każdego fragmentu można zastosować podobne podejście aż dojdzie się do tak małych fragmentów, że rozwiązanie zadania dla nich staje się niemal oczywiste. Technika ta prowadzi zwykle do powstania algorytmów rekurencyjnych. Samo pojęcie rekurencji i proste przykłady tego typu algorytmów przedstawiono między innymi w innym artykule opublikowanym w niniejszym czasopiśmie [9].

Technikę dziel i zwyciężaj z powodzeniem można zastosować w omawianym problemie. Na początek zostanie przyjęte, że tablica, w której poszukuje się indeksów najmniejszego i największego elementu ma rozmiar będący potęgą liczby 2. W końcowej wersji algorytmu, założenie to zostanie zignorowane i algorytm zostanie przedstawiony dla dowolnej wartości n. Jednak przyjęcie tego założenia znacznie ułatwia wyjaśnienie idei proponowanego rozwiązania.

Niech zatem będzie dana 16-elementowa tablica `a[]`, wypełniona liczbami rzeczywistymi. Zadanie polega na znalezieniu indeksów najmniejszego i największego elementu w tej tablicy. Rozwiązanie tego zadania jest nietrywialne, dlatego 16-elementową tablicę można podzielić na dwie podtablice liczące po 8 elementów. Pierwszą podtablicę tworzą elementy tablicy `a[]` o indeksach z zakresu od 0 do 7, a drugą elementy o indeksach od 8 do 15. Jeżeli w obu podtablicach zostaną znalezione indeksy najmniejszego i największego elementu, to aby znaleźć indeks największego elementu w całej 16-elementowej tablicy `a[]`, wystarczy porównać ze sobą największe elementy w obu podtablicach i wybrać ten z nich, który jest większy. Podobnie aby znaleźć indeks najmniejszego elementu w całej tablicy, wystarczy porównać elementy najmniejsze w obu podtablicach. Mechanizm podziału tablicy na coraz mniejsze fragmenty zilustrowano na rysunku 1.



Rysunek 1. Podział tablicy na coraz mniejsze fragmenty

Tylko jak znaleźć indeksy największego i najmniejszego elementu w obu 8-elementowych podtablicach? Osiem elementów to wciąż za dużo, dlatego też każda 8-elementowa tablica może zostać podzielona na dwie podtablice 4-elementowe. Każda tablica 4-elementowa może zostać z kolei podzielona na dwie tablice 2-elementowe. W przypadku tablic liczących po 2 elementy o różnych wartościach, by znaleźć indeks najmniejszego i największego elementu, wystarczy te dwa elementy ze sobą porównać. Tym samym w przypadku tablic 2-elementowych wystarcza jedno porównanie. Tablice o większej liczbie elementów są dzielone na dwie połowy, poszukuje się indeksów największego i najmniejszego elementu w każdej połowie z osobna, a następnie porównuje ze sobą elementy największe z obu połówek i elementy najmniejsze z obu połówek. Tym samym dla każdej z tablic liczących więcej niż 2 elementy (po ich podzieleniu i znalezieniu "ekstremów lokalnych") do znalezienia wyniku potrzebne są po dwa dodatkowe porównania.

Aby wyznaczyć łączną liczbę porównań można zauważyć, że na każdym poziomie podziału na podtablice, za wyjątkiem ostatniego (czyli tego, gdzie mamy do czynienia z fragmentami 2-elementowymi), wykonywane są po 2 porównania dla każdej tablicy z osobna. Na ostatnim poziomie dla każdej pary elementów wystarcza jedno porównanie. Łącznie na ostatnim poziomie jest $\frac{n}{2}$ tablic 2-elementowych, zaś liczba[1] wszystkich uwzględnianych podtablic (o różnych rozmiarach) wynosi $n-1$. Zatem wzór określający łączną liczbę porównań ma postać

$$2(n-1) - \frac{n}{2} = \frac{3}{2}n - 2. \tag{6}$$

Wcześniej prezentowane rozwiązania miały złożoność rzędu $2n$. Zastosowanie metody dziel i zwyciężaj, gdzie złożoność jest rzędu $\frac{3}{2}n$, pozwoliło zatem uzyskać algorytm o złożoności około 25% lepszej niż algorytmy poprzednio rozważane. Algorytm ów w pełnej postaci przedstawiono na listingu 5. Funkcja `minmax4` opisuje sposób poszukiwania indeksów elementów najmniejszego i największego we fragmencie tablicy `a[]` rozpoczynającym się od elementu o indeksie `lewy` i kończącym na elemencie o indeksie `prawy`. Wynik działania funkcji, czyli indeksy najmniejszego i największego elementu w rozpatrywanym fragmencie, zwracany jest poprzez parametry `i_min` i `i_max`. Jeżeli fragment ten składa się z co najmniej 3 elementów, wykonywany jest kod zawarty w liniach 16-29. Taki fragment tablicy dzielony jest na dwie połówki, w każdej z nich poszukuje się indeksów najmniejszego i największego elementu (linie 19 i 20), następnie porównuje się parami elementy najmniejsze z obu połówek i określa indeks elementu najmniejszego w całym rozpatrywanym fragmencie (linie 21-24), po czym robi się to samo w poszukiwaniu elementu największego w tym fragmencie tablicy (linie 25-28). Gdy fragment tablicy składa się z dwóch elementów (linie 5-15), do znalezienia indeksów elementów najmniejszego i największego wystarcza jedno porównanie (linia 6). Wreszcie, gdy rozważany fragment tablicy składa się tylko z jednego elementu (linie 3 i 4), żadne porównanie nie jest potrzebne, ten element jest jednocześnie najmniejszym i największym.

```
1  void minmax4( double a[], int lewy, int prawy, int &i_min, int &i_max )
2  {
3      if (lewy == prawy) // jednoelementowy fragment tablicy
4          i_min = i_max = lewy;
5      else if (lewy + 1 == prawy) // tablica 2-elementowa
6          if (a[lewy] < a[prawy])
7          {
8              i_min = lewy;
9              i_max = prawy;
10         }
11         else
12         {
13             i_min = prawy;
```

[1] Wynika to ze wzoru na sumę częściową ciągu geometrycznego, przy założeniu, że n jest potęgą 2. Liczba ta bowiem, to suma: $1+2+4+\ldots+\frac{n}{2}$.

```
14              i_max = lewy;
15          }
16      else // fragment liczy więcej niż 2 elementy
17      {
18          int i_min1, i_max1, i_min2, i_max2;
19          minmax4(a, lewy, (lewy + prawy) / 2, i_min1, i_max1);
20          minmax4(a, (lewy + prawy) / 2 + 1, prawy, i_min2, i_max2);
21          if (a[i_min1] < a[i_min2])
22              i_min = i_min1;
23          else
24              i_min = i_min2;
25          if (a[i_max1] > a[i_max2])
26              i_max = i_max1;
27          else
28              i_max = i_max2;
29      }
30  }
```

Listing 5. Algorytm rekurencyjnego poszukiwania indeksów najmniejszego i największego elementów w tablicy

Przedstawiony na listingu 5 algorytm charakteryzuje się najlepszą złożonością wśród algorytmów realizowanych sekwencyjnie, czyli takich, gdzie oparty o dany algorytm program wykonywany jest na jednym procesorze i w danym momencie wykonywana jest tylko jedna instrukcja. Należy jednak zauważyć, że podczas określania złożoności algorytmu z listingu 5 pominięto całkowicie czas związany z wywoływaniem kolejnych instancji funkcji `minmax4`. W praktyce czas ten jest często większy niż czas poświęcony na realizację operacji dominującej (czyli porównania elementów tablicy), co niweczy zyski związane ze zmniejszeniem liczby porównań. W przypadku, gdyby jednak porównanie elementów tablicy było procesem bardziej złożonym (np. porównywano by napisy) zaobserwowano by skrócenie czasu wykonywania całego zadania w stosunku do algorytmów wcześniej prezentowanych.

5. Wersja równoległa

Często stosowaną metodą skrócenia czasu wykonywania algorytmów, jest zwiększenie liczby procesorów zaangażowanych w realizację zadania. W dalszej części rozważań założono, że dostępnych jest zawsze wystarczająco dużo procesorów. Aby jednak z tych procesorów skorzystać konieczne jest dostosowanie algorytmu do realizacji w sposób równoległy. Nie każdy algorytm łatwo zrównoleglić. Wśród wcześniej przedstawionych algorytmów sekwencyjnych tylko ostatni jest dobrym kandydatem do stworzenia jego wersji równoległej. Jeżeli dostępnych jest co najmniej $\frac{n}{2}$ procesorów, wszystkie podtablice na każdym poziomie wywołań rekurencyjnych można przetwarzać równolegle, co prowadzi do algorytmu przedstawionego na listingu 6.

```
1  void minmax5( double a[], int lewy, int prawy, int &i_min, int &i_max )
2  {
3      if (lewy == prawy) // jednoelementowy fragment tablicy
4          i_min = i_max = lewy;
5      else if (lewy + 1 == prawy) // tablica 2-elementowa
6          if (a[lewy] < a[prawy])
7          {
8              i_min = lewy;
9              i_max = prawy;
10         }
11         else
```

```
12          {
13              i_min = prawy;
14              i_max = lewy;
15          }
16      else // fragment liczy więcej niż 2 elementy
17      {
18          int i_min1, i_max1, i_min2, i_max2;
19          parallel
20          {
21              minmax5(a, lewy, (lewy + prawy) / 2, i_min1, i_max1);
22              minmax5(a, (lewy + prawy) / 2 + 1, prawy, i_min2, i_max2);
23          }
24          if (a[i_min1] < a[i_min2])
25              min = min1;
26          else
27              i_min = i_min2;
28          if (a[i_max1] > a[i_max2])
29              i_max = i_max1;
30          else
31              i_max = i_max2;
32      }
33  }
```

Listing 6. Algorytm równoległy rekurencyjnego poszukiwania indeksów najmniejszego i największego elementów w tablicy

Uważny Czytelnik spostrzeże, że w stosunku do algorytmu sekwencyjnego wprowadzono tu tylko jedną zmianę. Wywołania funkcji `minmax4` umieszczone w liniach 19 i 20 na listingu 5, w wersji równoległej zostały otoczone dodatkową instrukcją `parallel` (linie 19-23). Instrukcja taka nie występuje w języku C, została wprowadzona wyłącznie na potrzeby niniejszego opracowania, aby wyrazić fakt, że wszystkie instrukcje otoczone nawiasami klamrowymi po słowie `parallel` wykonywane są równolegle, w tym samym czasie przez różne procesory. A zatem w czasie, gdy jeden z procesorów wykonuje instrukcję z linii 21, inny procesor wykonuje instrukcję z linii 22. Wtedy wszystkie operacje wykonywane na tym samym poziomie wywołań rekurencyjnych można wykonać równolegle przez różne procesory. Można zauważyć, że na ostatnim poziomie wywołań (dla dwuelementowego fragmentu tablicy) wystarcza jedna operacja porównania elementów tablicy. Na każdym wcześniejszym etapie potrzebne są po dwie operacje dominujące (jedna by znaleźć indeks elementu najmniejszego, druga - największego). W przypadku, gdy $n = 2^k$ gdzie $k \in \mathbb{N}$, przetwarzanie rekurencyjne realizowane jest na k poziomach i w efekcie uzyskuje się złożoność postaci $2\log_2 n - 1$ porównań. Jeżeli n nie jest potęgą dwójki, wartość $\log_2 n$ musi zostać zastąpiona jej zaokrągleniem do góry do najbliższej liczby całkowitej, tym samym wzór określający liczbę porównań zmienia się do postaci $2\lceil \log_2 n \rceil - 1$

Dzięki zastosowaniu przetwarzania równoległego udało się znacząco poprawić złożoność algorytmu, zmniejszając złożoność obliczeniową z liniowej w wersji z jednym procesorem do złożoności logarytmicznej, gdy dostępnych jest $\frac{n}{2}$ procesorów. W kolejnym rozdziale zostanie zaprezentowany algorytm o złożoności stałej.

6. Algorytm o stałym czasie wykonania

W tym przypadku należy przyjąć, że dostępnych jest n^2 procesorów. Procesory te należy rozmieścić w postaci 2-wymiarowej macierzy o wymiarach $n \times n$ (z indeksami od 0 do n-1). Symbol P_{ij} reprezentować

będzie procesor w i-tym wierszu i j-tej kolumnie. Algorytm wykorzystuje ponadto dwie jednowymiarowe tablice o długości n, których elementami są wartości logiczne (`true` i `false`). Tablica `tmax[]` odpowiada elementowi maksymalnemu, `tmin[]` - elementowi minimalnemu. I tak, jeśli `tmax[i]=false` oznacza to, że element `a[i]` nie jest na pewno elementem maksymalnym. Jeśli `tmax[i]=true` to znaczy, że element `a[i]` może być elementem maksymalnym. Podobnie dla elementu minimalnego i tablicy `tmin[]`. Zakładamy, że możliwy jest równoczesny zapis elementów w tablicach `tmin[]` i `tmax[]` przez wiele procesorów, ponieważ wszystkie te procesory będą wpisywać tę samą wartość dla tego samego elementu.

Aby poprawnie zapisać algorytm konieczne jest wprowadzenie jeszcze jednej instrukcji przetwarzania równoległego, niedostępnej w języku C. Tą instrukcją jest `parallel for`. Przyjęto, że instrukcja ta modyfikuje klasyczną instrukcję `for` w taki sposób, że każdy przebieg pętli jest realizowany przez oddzielny procesor równolegle z pozostałymi przebiegami, wykonywanymi przez inne procesory. Algorytm wykonujący całe zadanie w stałym czasie, niezależnie od rozmiaru tablicy, przedstawia listing 7.

```
1  void minmax6( double a[], int n, int &i_min, int &i_max )
2  {
3      bool tmin[n], tmax[n];
4      // inicjalizacja zmiennych roboczych
5      parallel for (int i=0; i<n; i++)
6          tmin[i] = tmax[i] = true;
7      parallel for (int i=0; i<n; i++)
8          parallel for (int j=0; j<N; j++)
9          {
10             if (a[i] < a[j])
11                 tmax[i] = false;
12             if (a[i] > a[j])
13                 tmin[i] = false;
14         }
15     // synchronizacja pracy procesorów
16     // spośród wszystkich kandydatów na minimum i maksimum wybieram te,
17     // które mają najmniejszy indeks (dotyczy przypadku, gdy są powtórzenia)
18     parallel for (int i=0; i<n; i++)
19         parallel for (int j=0; j<n; j++)
20             if (tmin[i] && i<j)
21                 tmin[j] = false;
22             if (tmax[i] && i<j)
23                 tmax[j] = false;
24     // na pewno tylko po jednym elemencie tablic tmin[] i tmax[] zawiera true
25     parallel for (int i=0; i<n; i++)
26     {
27         if (tmin[i])
28             i_min=i;
29         if (tmax[i])
30             i_max=i;
31     }
32 }
```

Listing 7. Równoległa realizacja zadania poszukiwania indeksów najmniejszego i największego elementów w tablicy w stałym czasie

Zadanie znalezienia indeksów minimum i maksimum w tablicy `a[]` realizowane jest w kilku etapach. Najpierw (wiersze 5 i 6) ma miejsce inicjalizacja tablic `tmin[]` i `tmax[]`. Uczestniczą w niej procesory P_{i0} z górnego wiersza macierzy procesorów. Każdy z nich jednocześnie wpisuje wartość `true` do odpowiednich elementów obydwu inicjalizowanych tablic. Wpisanie wartości `true` oznacza, że odpowiedni element w tablicy `a[]` może być elementem najmniejszym (w przypadku `tmin[]`) lub największym (`tmax[]`) w całym zbiorze.

Kolejny etap (linie 7-14) realizują wszystkie dostępne procesory. Każdy z procesorów P_{ij} równolegle porównuje ze sobą dwa elementy `a[i]` i `a[j]`. Na podstawie wyniku porównania wnioskuje, czy element `a[i]` może być elementem minimalnym bądź maksymalnym. Jeżeli w wyniku porównania element i-ty okazuje się być mniejszym niż j-ty, oczywistym jest, że element i-ty nie może być największym elementem w tablicy `a[]`. Stąd też w takim przypadku w odpowiednie miejsce tablicy `tmax[]` wpisywana jest wartość `false`. Podobnie, jeśli element `a[i]` okaże się większy niż `a[j]`, element i-ty nie może być najmniejszym, zatem do tablicy `tmin[]` w odpowiednim miejscu wpisywane jest `false`.

Jeżeli w tablicy `a[]` nie ma powtórzeń tych samych wartości, po wykonaniu tego kroku w tablicach `tmin[]` i `tmax[]` wartości `true` będą występować dokładnie tylko jeden raz. Gdyby jednak w tablicy `a[]` wartość najmniejsza lub największa powtarzała się kilkakrotnie, w odpowiedniej tablicy (`tmin[]` lub `tmax[]`) kilkakrotnie pozostanie wpisana wartość `true`. W takim przypadku konieczne jest wykonanie kolejnego etapu (linie 18-23), dzięki któremu wartość `true` pozostanie tylko w jednym elemencie wspomnianych tablic (dokładniej elemencie o najmniejszym indeksie spośród wszystkich zawierających wcześniej `true`). Na tym etapie znów wykorzystywane są wszystkie procesory.

Ostatnim krokiem (linie 25-31) jest równoległe przeszukanie tablic `tmin[]` i `tmax[]` przez procesory z pierwszego wiersza P_{i0} i wpisanie wartości indeksu i jako wyniku do odpowiednio zmiennej `i_min` (dla tablicy `tmin[]`) i `i_max` (dla tablicy `tmax[]`).

Dla tego algorytmu operacje dominujące (linie 10 i 12) są wykonywane w czasie stałym niezależnie od rozmiaru danych n. Udało się to osiągnąć dzięki wykorzystaniu n^2 procesorów. W praktyce można zmniejszyć liczbę faktycznie użytych procesorów do $(n^2-n)/2+1$ (wykorzystać tylko procesory powyżej głównej przekątnej macierzy oraz jeden dodatkowy w pierwszym wierszu macierzy).

7. Podsumowanie

W artykule na przykładzie jednego prostego problemu zaprezentowano, jak wyznaczać złożoność obliczeniową algorytmów oraz jak można optymalizować algorytmy pod względem czasu ich realizacji. Pokazano także, że nie wszystkie algorytmy łatwo można zrównoleglić. Aby uzyskać rzeczywistą poprawę złożoności algorytmu, często należy znacząco zmienić sposób patrzenia na rozwiązywany problem (co szczególnie widać na przykładzie algorytmu rozwiązującego postawione zadanie w stałym czasie).

Niniejsze opracowanie jest próbą pokazania, czym jest efektywność czasowa algorytmów i jak można ją poprawiać. Aby to uczynić, posłużono się stosunkowo prostym problemem. Czytelnik pragnący poszerzyć swoją wiedzę może sięgnąć do innych opracowań, dostępnych tak w postaci drukowanej [2], [3] jak i elektronicznej [1], [7].

Literatura

1. Avocado Software, *Analiza sprawności algorytmów*, `xion.org.pl/files/texts/mgt/pdf/M_B.pdf`. [Dostęp 15.09.2021]
2. L. Banachowski, K. Diks, W. Rytter, *Algorytmy i struktury danych*, Wydawnictwa Naukowo-Techniczne, Warszawa 2006.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Wprowadzenie do algorytmów*, Wydawnictwa Naukowo-Techniczne, Warszawa 2004.

4. *Kurs C - programowanie*, Kanał o wszystkim, https://www.youtube.com/playlist?list=PL6aekdNhY7DBvSnKOHUUBb-OH4y41HoZw [Dostęp 15.09.2021]
5. M. Piaszczak, *Kurs języka C*, http://kurs-c.manifo.com/ [Dostęp 15.09.2021]
6. *Przekazywanie parametru przez wartość i referencję*, https://4programmers.net/C/Przekazywanie_parametru_przez_wartość_i_referencję. [Dostęp 15.09.2021]
7. *Samouczek programisty*, https://www.samouczekprogramisty.pl/podstawy-zlozonosci-obliczeniowej/. [Dostęp 15.09.2021]
8. *Tablica jako argument funkcji*, http://www.algorytm.edu.pl/tablice-w-c/tablica-jako-argument-funkcji.html. [Dostęp 15.09.2021]
9. R. Tutajewicz, *Wprowadzenie do algorytmów rekurencyjnych*, MINUT 2021 (3), 87-97.
10. E. W. Weisstein, *Euler-Mascheroni Constant*, MathWorld - A Wolfram Web Resource, https://mathworld.wolfram.com/Euler-MascheroniConstant.html. [Dostęp 15.09.2021]