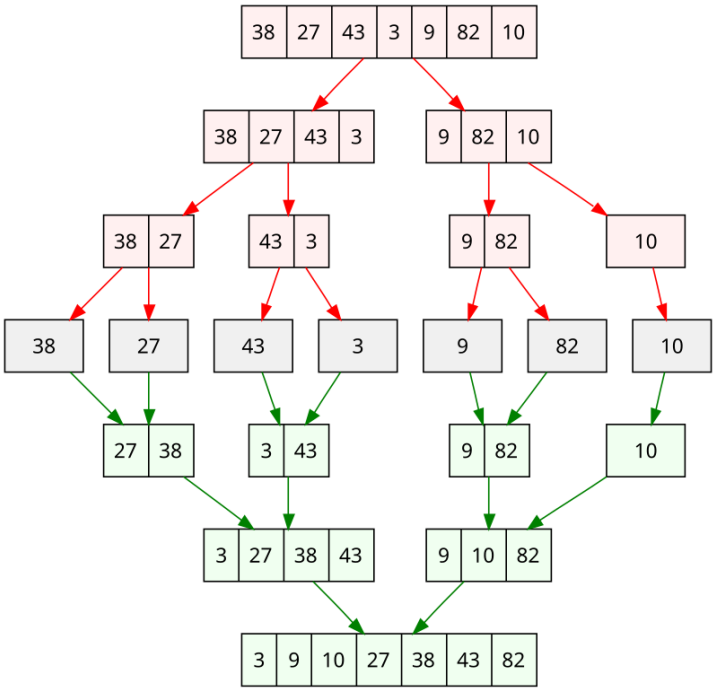
Dziel i zwyciężaj to strategia w programowaniu i algorytmice, która oznacza „Dziel i rządź”. Jej istota to:

1. Podziel problem na mniejsze części.
2. Rozwiąż każdą z nich osobno.
3. Połącz wyniki w jedno rozwiązanie.

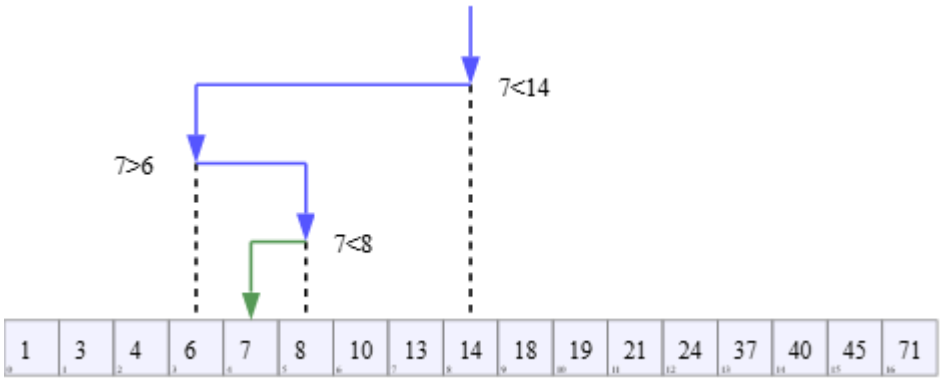


To podejście jest często używane razem z rekurencją

Wyszukiwanie binarne

Wyszukiwanie binarne to efektywna metoda znajdowania elementu w posortowanej tablicy. Zamiast sprawdzania każdego elementu po kolei, algorytm analizuje środkowy element i na jego podstawie dzieli tablicę na pół, kontynuując wyszukiwanie tylko w jednej z części.

Przykład działania:
Dla tablicy:
[1, 3, 5, 7, 9, 11]
celem jest znalezienie liczby **7.**



1. Środkowy element to **5.** Ponieważ **5 < 7**, wyszukiwanie przenosi się do prawej części.
2. Środek prawej części to **9.** Ponieważ **9 > 7,** wyszukiwanie przenosi się w lewo.
3. Znaleziono **7**.

Zaleta:
Zamiast sprawdzania wszystkich 6 elementów, algorytm potrzebował tylko 2–3 porównań. Dzięki temu działa dużo szybciej niż liniowe przeszukiwanie, szczególnie przy dużych zbiorach danych.

Ось переклад польською:

QuickSort to jeden z najszybszych sposobów sortowania tablicy (listy liczb).

Jak to działa:

Wybierasz pivot – to liczba odniesienia (punkt podziału).

1. Dzielisz wszystkie liczby na:
   - Te, które są mniejsze od pivotu.
   - Te, które są większe od pivotu.
2. Powtarzasz te same kroki dla każdej części (czyli znowu dzielisz i sortujesz).
3. Gdy wszystko jest posortowane – łączysz razem.
4.

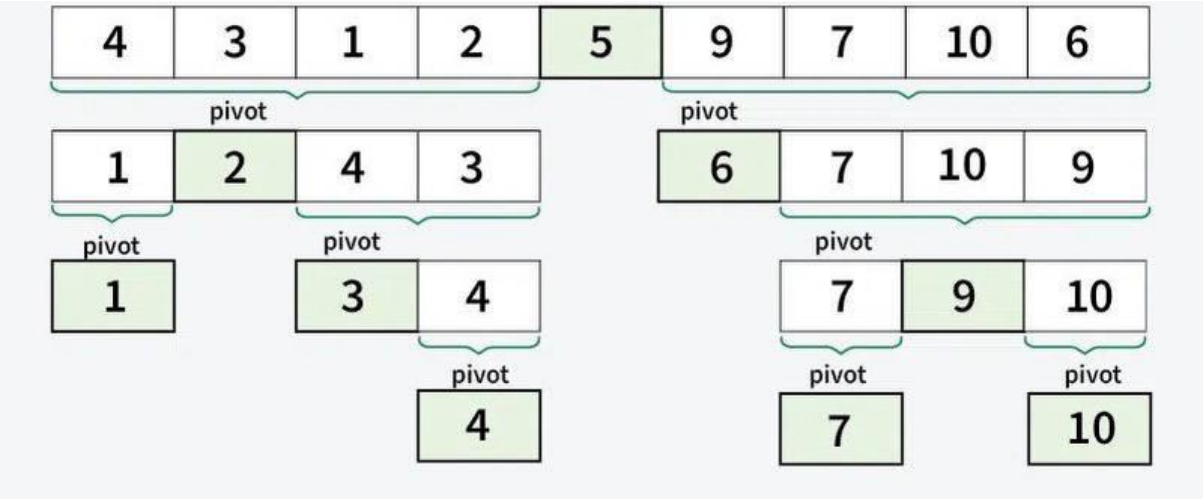
Przykład:

- [4, 3, 1, 2, 5, 9, 7, 10, 6]
  Pivot = 4
- mniejsze: [3, 1, 2], większe: [5, 9, 7, 10, 6]
- Sortujemy [3, 1, 2]

- pivot = 3
- mniejsze: [1, 2], większe: []
- sortujemy [1, 2] z pivotem = 1
- wynik: [1, 2, 3]
- Sortujemy [5, 9, 7, 10, 6]
- pivot = 5
- mniejsze: [], większe: [9, 7, 10, 6]
- pivot = 9
- mniejsze: [7, 6], większe: [10]
- sortujemy [7, 6] z pivotem = 7
- wynik: [6, 7]

  Wynik końcowy:

  [1, 2, 3, 4, 5, 6, 7, 9, 10]




```
void zamien(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
```

To znaczy:

- tworzymy funkcję, która zamienia dwa liczby miejscami
- oznacza, że pracujemy na oryginalnych zmiennych, a nie na ich kopiach
- Najpierw a zapisujemy w temp
- Potem a dostaje wartość b
- b dostaje wartość z temp

C++

```cpp
int podziel(int tablica[], int lewy, int prawy) {
    int pivot = tablica[prawy];
    int i = lewy - 1;
    for (int j = lewy; j < prawy; j++) {
        if (tablica[j] < pivot) {
            i++;
            zamien(tablica[i], tablica[j]);
        }
    }
    zamien(tablica[i + 1], tablica[prawy]);
    return i + 1;
```

- to funkcja, która dzieli tablicę w algorytmie QuickSort
- Pivot to liczba odniesienia (tutaj: ostatni element)
- liczby mniejsze od pivotu przesuwane są w lewo
- na końcu ustawiamy pivot na swoje miejsce
- funkcja zwraca pozycję, gdzie pivot się znalazł

```cpp
void quickSort(int tablica[], int lewy, int prawy) {
    if (lewy < prawy) {
        int pi = podziel(tablica, lewy, prawy);
        quickSort(tablica, lewy, pi - 1);
        quickSort(tablica, pi + 1, prawy);
    }
```

- główna funkcja sortowania szybkim algorytmem
- jeśli lewa część jest mniejsza od prawej:
- dzielimy tablicę na dwie części
- sortujemy osobno lewą i prawą stronę

```cpp
int wyszukiwanieBinarne(int tablica[], int lewy, int prawy, int liczba) {
    if (lewy <= prawy) {
        int srodek = lewy + (prawy - lewy) / 2;
        if (tablica[srodek] == liczba)
            return srodek;
        else if (tablica[srodek] > liczba)
            return wyszukiwanieBinarne(tablica, lewy, srodek - 1, liczba);
        else
            return wyszukiwanieBinarne(tablica, srodek + 1, prawy, liczba);
    }
    return -1;
}
```

- funkcja do wyszukiwania binarnego
- dzieli tablicę na pół i sprawdza środek
- jeśli liczba jest znaleziona -> zwraca indeks
- jeśli nie -> przeszukuje lewą lub prawą część

- jeśli nie znajdzie nic -> zwraca -1

```
int n;
cout << "podaj dlugosc tablicy: ";
cin >> n;
```

- tworzymy zmienną n – to liczba elementów w tablicy
- pytamy użytkownika, ile liczb chce wpisać

```
int tablica[100];
```

- tworzymy tablicę, w której może być maksymalnie 100 liczb

```
cout << "wpisz " << n << " liczb:\n";
for (int i = 0; i < n; i++) {
    cin >> tablica[i];
}
```

- rosimy użytkownika, aby wpisał n liczb
- zapisujemy te liczby do tablicy

```
quickSort(tablica, 0, n - 1);
```

- uruchamiamy funkcję sortującą dla całej tablicy (od 0 do n-1).

```
cout << "posortowana tablica:\n";
for (int i = 0; i < n; i++) {
    cout << tablica[i] << " ";
}
cout << endl;
```

- wypisujemy na ekran posortowaną tablicę
- każda liczba jest pokazana obok siebie

```
int liczba;
cout << "wpisz liczbe do wyszukania: ";
cin >> liczba;
```

- tworzymy zmienną liczba, w której zapiszemy to, czego użytkownik chce szukać

pytamy o tę liczbę

```
int indeks = wyszukiwanieBinarne(tablica, 0, n - 1, liczba);
```

- uruchamiamy wyszukiwanie binarne

- zapisujemy wynik (indeks) do zmiennej index

```
if (indeks != -1)
    cout << "liczba znaleziona na indeksie: " << indeks << endl;
else
    cout << "liczba nie zostala znaleziona.\n";

return 0;
```

- jeśli indeks nie wynosi -1, to liczba została znaleziona
- jeśli wynik to -1, to liczba nie istnieje w tablicy
-

- dodano bibliotekę chrono

```
#include <chrono>
```

- Pozwala używać narzędzi do pomiaru czasu — np. high_resolution_clock.
- Bez tej biblioteki nie moglibyśmy sprawdzić, ile trwa wykonanie funkcji.

- dodano using namespace chrono;

```
using namespace chrono;
```

- Ułatwia zapis: zamiast pisać pełne nazwy

- pomiar czasu sortowania QuickSort

```
auto start_sort = high_resolution_clock::now();
quickSort(tablica, 0, n - 1);
auto end_sort = high_resolution_clock::now();
auto czas_sortowania = duration_cast<microseconds>(end_sort - start_sort).count();
```

1. start_sort - zapamiętuje dokładny moment przed sortowaniem.
2. quickSort(...) - sortuje tablicę.
3. end_sort - zapamiętuje moment po sortowaniu.
4. czas_sortowania - oblicza ile mikrosekund minęło między startem a końcem.

Dzięki temu wiemy, jak szybko działa algorytm sortowania

- pomiar czasu wyszukiwania binarnego

```
auto start_szukania = high_resolution_clock::now();
int indeks = wyszukiwanieBinarne(tablica, 0, n - 1, liczba);
auto end_szukania = high_resolution_clock::now();
auto czas_szukania = duration_cast<microseconds>(end_szukania - start_szukania).count();
```

1. start_szukania - zapamiętuje czas przed szukaniem.
2. wyszukiwanieBinarne(...) - wykonuje się normalnie.
3. end_szukania - zapisuje czas po szukaniu.
4. czas_szukania - mówi, ile mikrosekund trwało wyszukiwanie liczby.

Pozwala sprawdzić, jak szybko działa wyszukiwanie w posortowanej tablicy.

- dodano wypisywanie czasów

```
cout << "\nCzas sortowania: " << czas_sortowania << " mikrosekund\n";
cout << "Czas wyszukiwania: " << czas_szukania << " mikrosekund\n";
```

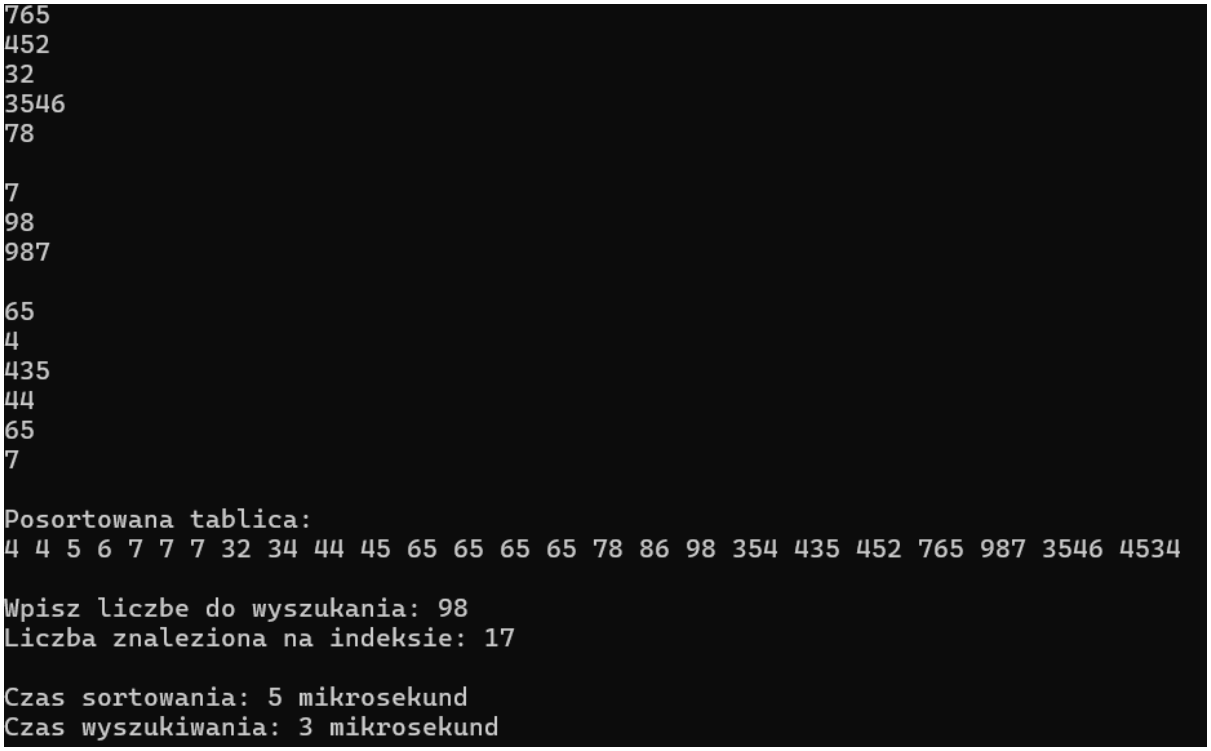
- Wypisuje użytkownikowi na ekran:
- ile czasu zajęło sortowanie
- ile czasu zajęło szukanie liczby

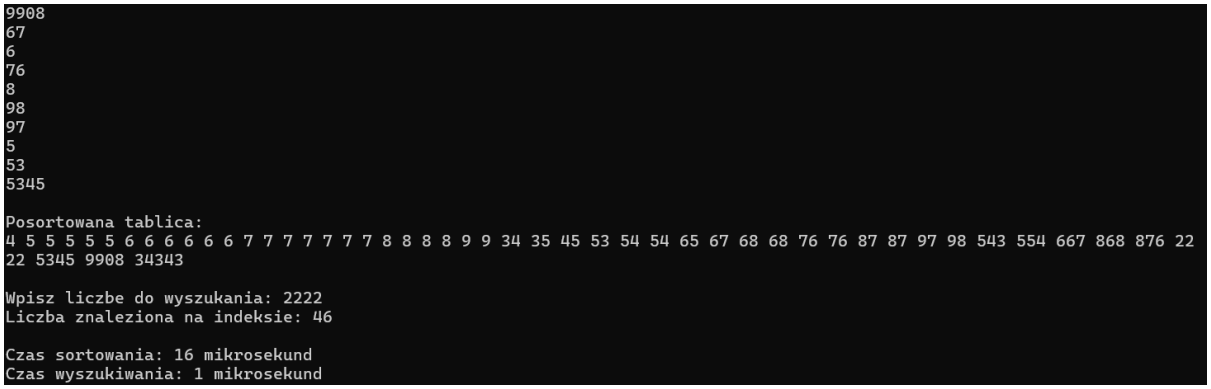
```
Podaj dlugosc tablicy: 10
Wpisz 10 liczb:
1
2
3
4
5
6
7
8
9
99

Posortowana tablica:
1 2 3 4 5 6 7 8 9 99

Wpisz liczbe do wyszukania: 9
Liczba znaleziona na indeksie: 8

Czas sortowania: 3 mikrosekund
Czas wyszukiwania: 1 mikrosekund
```





Liczba elementow	Czas sortowania	Czas wyszukiwania
10	3	1
25	5	3
50	16	1

1. Czas sortowania rośnie wraz ze wzrostem liczby elementów.
   Dla 10 elementów sortowanie trwało 3 mikrosekundy, a dla 50 już 16 mikrosekund.
2. Czas wyszukiwania binarnego pozostaje bardzo mały, nawet gdy liczba elementów rośnie.
   W przypadku 10, 25 i 50 elementów wynik był 1-3 mikrosekundy.
3. QuickSort działa szybko i skalowalnie, ale jego czas zależy od długości tablicy.

4. Wyszukiwanie binarne jest bardzo efektywne, bo sprawdza tylko kilka pozycji - niezależnie od rozmiaru danych.

- dodano licznik zamian

```
int licznikSwapow = 0;
```

- Zmienna zlicza, ile razy została wywołana funkcja zamien(...), czyli ile razy elementy tablicy były zamieniane miejscami.
- To pozwala określić, jak wiele operacji fizycznych wykonuje algorytm Quicksort - czyli ile "kosztuje" sortowanie w działaniach.

- dodano licznik wywołań funkcji binarnego wyszukiwania
  ```
  int licznikBinarnego = 0;
  ```
- Zmienna zlicza, ile razy została wywołana funkcja wyszukiwanieBinarne(...) wliczając wszystkie wywołania rekurencyjne.

- Pozwala przeanalizować, ile kroków potrzeba do znalezienia liczby — i jak to zależy od rozmiaru tablicy.

- inkrementacja w funkcji zamien()
  ```
  licznikSwapow++;
  ```

- Za każdym razem, gdy zamieniamy dwa elementy - dodaje 1 do licznika licznikSwapow

- inkrementacja w wyszukiwanieBinarne()

- ```
  licznikBinarnego++;
  ```
  Za każdym razem, gdy uruchamia się funkcja wyszukiwanieBinarne - zwiększa licznik.

- zwiększenie rozmiaru tablicy
  ```

- ```
  int tablica[1000000];
  ```
  Pozwala testować duże zestawy danych, zgodnie z poleceniem z instrukcji.

- wypisywanie dodatkowych wyników na końcu

- ```
  cout << "Liczba operacji zamiany (swap): " << licznikSwapow << endl;
  cout << "Liczba wywolan wyszukiwania binarnego: " << licznikBinarnego << endl;
  cout << "Czas laczny: " << (czas_sortowania + czas_szukania) << " mikrosekund\n";
  ```
- pokazuje łączną liczbę swapów,
- liczbę kroków binarnego wyszukiwania,
- oraz całkowity czas wykonania programu (sort + wyszukiwanie).

Wnioski

W programie mogą występować błędy związane z nieprawidłowym przetwarzaniem granic i warunków stosowanych w algorytmach sortowania i wyszukiwania. Może to prowadzić do błędnych wyników, a nawet do awarii działania programu. Wybór elementu pivot w algorytmie szybkiego sortowania również ma istotne znaczenie — stosowanie stałego podejścia obniża efektywność przy niektórych danych wejściowych, dlatego warto wdrożyć lepszą strategię. Program nie kontroluje liczby elementów wprowadzanych przez użytkownika, co może prowadzić do przekroczenia rozmiaru tablicy, dlatego należy zaimplementować sprawdzenie dopuszczalnego rozmiaru danych przed ich przetwarzaniem.

Funkcje rekurencyjne w kodzie działają bez jasno określonych warunków zakończenia, co stwarza ryzyko nadmiernych wywołań lub błędów wykonania. Ponadto zliczanie operacji (np. zamian) w statystykach nie zawsze odpowiada rzeczywistemu działaniu algorytmu, ponieważ liczniki są aktualizowane niezależnie od faktycznych operacji. Prowadzi to do zniekształconych wniosków analitycznych opartych na działaniu programu. Wprowadzanie liczb przez użytkownika również nie jest objęte kontrolą poprawności — brak weryfikacji typu lub formatu danych może prowadzić do pojawienia się nieprawidłowych danych w programie, co wpływa na jego stabilność i dokładność.

Aby zapewnić poprawność działania, wydajność i łatwość utrzymania kodu, warto ulepszyć logikę sprawdzania, zwiększyć odporność programu na błędy wejścia oraz doprecyzować sposób zbierania danych statystycznych. Wszystkie te działania przyczyniają się do zwiększenia niezawodności programu i lepszego zrozumienia jego działania zarówno przez użytkownika, jak i przez programistę.