Analiza Algorytmów Sortowania

Jan Sas 287299

18 października 2025

Opis Algorytmów i Wyników

1. Insertion Sort (Sortowanie przez wstawianie)

Zasada działania: Przechodzi przez tablicę, biorąc każdy element i wstawiając go na właściwe miejsce w już posortowanej części po lewej stronie.

Modyfikacja: Przetwarza elementy parami. Porównuje parę, a następnie wstawia najpierw większy, potem mniejszy element do posortowanej części.

Fragment kodu modyfikacji:

```
    // ... wewnątrz pętli for (i=1; i<n - 1; i+=2) ...
    int m_k, d_k; // mniejszy_klucz, duzy_klucz
    porownania++;
    if(A[i] > A[i+1]){ // Porownanie pary
      d_k = A[i]; m_k = A[i+1];
    } else {
      m_k = A[i]; d_k = A[i+1];
    }
    przypisania+=2; int j = i - 1;

    // Wstawianie wiekszego klucza (d_k)
    while(j >= 0){
      porownania++;
      if(A[j] > d_k){
        A[j + 2] = A[j]; j--; przypisania++;
      } else { break; }
    }
    A[j + 2] = d_k; przypisania++;

    // Wstawianie mniejszego klucza (m_k)
    while(j >= 0){
      porownania++;
      if (A[j] > m_k) {
        A[j + 1] = A[j]; j--; przypisania++;
      } else { break; }
    }
    A[j + 1] = m_k; przypisania++;
    // ...
```

Listing 1: Fragment modyfikacji Insertion Sort - obsługa pary

2. Merge Sort (Sortowanie przez scalanie)

Zasada działania: Dzieli tablicę na pół, rekurencyjnie sortuje połówki, a następnie scala je w jedną posortowaną całość.

Modyfikacja: Dzieli tablicę na **trzy** części, rekurencyjnie je sortuje, a potem scala wszystkie trzy naraz.

Fragment kodu modyfikacji:

```
1  void mod_merge_s(int A[], int p, int s1, int s2, int k, /*
       ...liczniki... */){
2    // ... (kopiowanie do L, S, P) ...
3    int i = 0, j = 0, l = 0, w = p;
4    while(i < r_L && j < r_S && l < r_P){ // Glowna petla
       scalajaca 3
5      przypisania++; porownania++;
6      if(L[i] <= S[j]){
7        porownania++;
8        if(L[i] <= P[l]){ A[w++] = L[i++]; }
9        else{ A[w++] = P[l++]; }
10     } else {
11       porownania++;
12       if(S[j] <= P[l]){ A[w++] = S[j++]; }
13       else{ A[w++] = P[l++]; }
14     }
15   }
16   // ... (petle czyszczace dla pozostalych 2 lub 1 tablicy)
       ...
17 }
18
19 void mod_merge_sort(int A[], int p, int k, /* ...liczniki...
     */){
20   if (k - p < 2) { /* ... warunek bazowy ... */ return; }
21   int s1 = p + (k - p) / 3; int s2 = p + 2 * (k - p) / 3; //
       Dwa punkty podzialu
22   mod_merge_sort(A, p, s1, /*...*/);
23   mod_merge_sort(A, s1 + 1, s2, /*...*/);
24   mod_merge_sort(A, s2 + 1, k, /*...*/);
25   mod_merge_s(A, p, s1, s2, k, /*...*/); // Scalanie trzech
26 }
```

Listing 2: Fragment modyfikacji Merge Sort - scalanie 3 części

3. Heap Sort (Sortowanie przez kopcowanie)

Zasada działania: Buduje z tablicy strukturę kopca (najczęściej binarnego typu max-heap), a następnie wielokrotnie usuwa największy element (korzeń), umieszcza go na końcu i naprawia kopiec.

Modyfikacja: Używa **kopca ternarnego** (trójkowego), gdzie rodzic ma do trojga dzieci. Wymaga to innych wzorów na indeksy dzieci i zmodyfikowanej funkcji `heapify`.

Fragment kodu modyfikacji:

```
void mod_heapify(int A[], int n, int i, /* ...liczniki... */) {
  int d = i;
  int c1 = 3 * i + 1; int c2 = 3 * i + 2; int c3 = 3 * i + 3; // Indeksy 3 dzieci

  // Znajdowanie najwiekszego sposrod 4 elementow (rodzic + 3 dzieci)
  if(c1 < n){ porownania++; if(A[c1] > A[d]){ d = c1; } }
  if(c2 < n){ porownania++; if(A[c2] > A[d]){ d = c2; } }
  if(c3 < n){ porownania++; if(A[c3] > A[d]){ d = c3; } }

  if(d != i){
    swap(A[i], A[d]); przypisania += 3;
    mod_heapify(A, n, d, /*...*/);
  }
}

void mod_heap_sort(int A[], int n, /* ...liczniki... */) {
  // ... (zerowanie licznikow) ...
  // Budowanie kopca ternarnego - inny indeks startowy
  for(int i=(n - 2)/3; i>=0; i--) { mod_heapify(A, n, i, /*...*/); }
  // Petla sortujaca
  for(int i=n - 1; i>0; i--) {
    swap(A[0], A[i]); przypisania += 3;
    mod_heapify(A, i, 0, /*...*/);
  }
}
```

Listing 3: Fragment modyfikacji Heap Sort - kopiec ternarny

4. Funkcja pomocnicza `wypisz`

Prosta funkcja iterująca przez tablicę i wypisująca jej elementy na konsolę, oddzielone spacjami.

```
1     void wypisz(int A[], int n){
2       for(int i=0; i<n; i++){
3         cout << A[i] << " ";
4       }
5       cout << "\n";
6     }
```

Listing 4: Funkcja wypisz

5. Funkcja testująca `main`

Główna funkcja programu odpowiedzialna za przeprowadzenie testów i pomiar wydajności. Jej działanie obejmuje: zdefiniowanie testowanych rozmiarów tablic (`n`); iterację przez te rozmiary; dla każdego `n` dynamiczne alokowanie pamięci na tablicę oryginalną i tymczasową; wypełnienie tablicy oryginalnej losowymi danymi; następnie dla każdego z sześciu algorytmów: skopiowanie danych do tablicy tymczasowej, zmierzenie czasu wykonania (za pomocą `<chrono>`), wywołanie funkcji sortującej (przekazując referencje do liczników), obliczenie i wypisanie czasu oraz liczby porównań i przypisań; na koniec zwolnienie zaalokowanej dynamicznie pamięci. Użycie `copy` przed każdym testem gwarantuje, że algorytmy działają na identycznych danych wejściowych.

```

    using namespace std::chrono;

    int main(){
      vector<int> n_values = {100, 1000, 10000, 50000};
      // ... (ustawienia generatora) ...
      cout << fixed << setprecision(3);

      for (int n : n_values) {
        // ... (alokacja, generowanie danych) ...

        // Przyklad testu dla Insertion Sort
        copy(original_data, original_data + n, temp_data);
        auto start = high_resolution_clock::now(); // Czas start
        insertion_sort(n, temp_data, porownania, przypisania);
        auto end = high_resolution_clock::now();   // Czas stop
        duration<double, milli> elapsed_ms = end - start; //
           Obliczenie roznicy

        cout << "Insertion Sort: \t" << porownania << " por, "
        << przypisania << " przy, " << elapsed_ms.count() << "
           ms" << endl;

        // ... (testy dla reszty algorytmow) ...

        delete[] original_data; // Zwolnienie pamieci
        delete[] temp_data;
      }
      return 0;
    }
```

Listing 5: Fragment funkcji main - testowanie i pomiar czasu

6. Tabela Wyników

Poniższa tabela przedstawia zebrane wyniki liczby porównań kluczy, przypisań kluczy oraz czasu wykonania (w milisekundach) dla poszczególnych algorytmów i różnych rozmiarów danych wejściowych (`n`).

Tabela 1: Porównanie wydajności algorytmów sortowania

Rozmiar (n)	Algorytm	Porównania	Przypisania	Czas (ms)
100	Insertion Sort	2,582	2,685	0.016
	Mod Insertion Sort	1,884	1,939	0.012
	Merge Sort	544	1,344	0.018
	Mod Merge Sort	668	876	0.018
	Heap Sort	1,030	1,758	0.017
	Mod Heap Sort	1,021	1,275	0.018
1,000	Insertion Sort	244,861	245,862	1.293
	Mod Insertion Sort	165,917	166,419	0.915
	Merge Sort	8,702	19,952	0.199
	Mod Merge Sort	10,756	13,084	0.174
	Heap Sort	16,860	27,261	0.251
	Mod Heap Sort	16,404	18,810	0.222
10,000	Insertion Sort	24,758,602	24,768,608	130.506
	Mod Insertion Sort	16,531,497	16,536,504	84.347
	Merge Sort	120,437	267,232	2.962
	Mod Merge Sort	149,280	173,756	2.037
	Heap Sort	235,448	372,786	3.244
	Mod Heap Sort	226,658	250,617	3.119
50,000	Insertion Sort	624,462,897	624,512,907	3510.284
	Mod Insertion Sort	415,981,205	416,006,216	2300.423
	Merge Sort	718,272	1,568,928	14.800
	Mod Merge Sort	893,789	1,000,000	11.351
	Heap Sort	1,409,992	2,213,046	21.060
	Mod Heap Sort	1,359,081	1,478,568	19.292

7. Omówienie Wyników

Tabela potwierdza, że algorytmy $O(n \log n)$ (Merge, Heap Sort) są znacznie wydajniejsze dla dużych n niż $O(n^2)$ (Insertion Sort). Modyfikacje wprowadzają kompromisy: Mod Insertion Sort jest szybszy; Mod Merge Sort wykonuje więcej porównań, ale mniej przypisań (i działa szybciej dla dużych n); Mod Heap Sort jest nieznacznie szybszy. Zmodyfikowany Merge Sort okazał się najszybszy w testach dla $n \geq 10000$.

8. Wnioski

- Złożoność asymptotyczna jest kluczowa dla dużych danych.
- Modyfikacje algorytmów mogą optymalizować liczbę porównań lub przypisań, co wpływa na rzeczywisty czas wykonania. Koszt operacji pamięci jest istotny.
- W przeprowadzonych testach zmodyfikowany Merge Sort (3-częściowy) okazał się najszybszy dla dużych zbiorów danych, mimo większej liczby porównań na krok.
- Wybór algorytmu zależy od rozmiaru danych i priorytetów (czas vs. liczba konkretnych operacji).