Sprawozdanie z Listy 1

Maciej Pluta
287339

1 Wprowadzenie

Celem mojej pracy było przetestowanie trzech klasycznych algorytmów sortowania: `Insertion Sort`, `Merge Sort` oraz `Heap Sort`, wraz z ich zmodyfikowanymi wersjami (wstawianie podwójne, dzielenie na trzy, kopiec ternarny). Najważniejszym zadaniem było dodanie liczników, które zliczałyby dokładnie to, co jest kluczowe w algorytmach: porównania oraz przypisania. Dzięki tym danym mogłem sprawdzić, czy wprowadzone przeze mnie modyfikacje faktycznie poprawiły wydajność sortowania.

2 Implementacja i Kluczowe Fragmenty Kodu

Wszystkie algorytmy zostały zaimplementowane jako osobne funkcje, a liczniki operacji były zerowane przed każdym testem w pliku main.cpp. Poniżej przedstawiam kluczowe fragmenty kodu modyfikacji, które pokazują, w którym miejscu zliczamy operacje.

2.1 Insertion Sort: Wstawianie podwójne (`insertionSortDouble`)

Jest to modyfikacja algorytmu sortowania przez wstawianie. W tej wersji w pętli głównej while sortujemy dwa elementy naraz. Zysk na wydajności wynika z drastycznego zmniejszenia kosztownych przypisań. Poniższy fragment kodu z pliku `insertion_sort.cpp` pokazuje kluczowy moment wstawiania dwóch elementów do już posortowanej części tablicy.

```
1         c.comparisons++;
2         if(arr[i] > arr[i+1]) { swap(arr[i], arr[i+1]); c.assignments
              += 3; }
3         int key1 = arr[i], key2 = arr[i+1]; c.assignments += 2;
4         int j = i - 1;
5         while(j >= 0 && arr[j] > key2) { c.comparisons++; arr[j+2] =
              arr[j]; c.assignments++; j--; }
6         if(j>=0) c.comparisons++;
7         arr[j+1] = key1; arr[j+2] = key2; c.assignments += 2;
```

Listing 1: Wstawianie podwójne: fragment kluczowego bloku wstawiania

2.2 Merge Sort: Dzielenie na trzy części (`merge3`)

W standardowym `Merge Sort` tablica jest dzielona na dwie połowy. Tutaj dzielimy ją na trzy części($\log_3 N$ zamiast $\log_2 N$). Kosztem tej modyfikacji jest to, że w procedurze scalania (`merge3`) musimy wykonać więcej porównań, aby znaleźć najmniejszy element spośród trzech grup. Poniższy fragment pokazuje rdzeń pętli scalającej.

```
while(i<=mid1 || j<=mid2 || k<=right){
    int val = INT_MAX;
    if(i<=mid1) val = arr[i];

    if(j<=mid2 && arr[j]<val){ val = arr[j]; c.comparisons++; } else
        if(j<=mid2) c.comparisons++;
    if(k<=right && arr[k]<val){ val = arr[k]; c.comparisons++; } else
        if(k<=right) c.comparisons++;

    temp.push_back(val); c.assignments++;
}
```

Listing 2: Merge Sort 3: fragment pętli scalania trzech podtablic

2.3 Heap Sort: Kopiec Ternarny (`heapSortTernarny`)

Zwykły `Heap Sort` używa kopca binarnego. My używamy kopca ternarnego (podstawa 3), co prowadzi do mniejszej wysokości kopca i krótszej drogi, jaką elementy muszą pokonać w dół w funkcji `heapify`. Ten zysk musi zrównoważyć koszt trzech porównań (zamiast dwóch) w każdym kroku. Poniższy kod jest całym ciągłym fragmentem funkcji `heapifyTernary` z pliku `heap_sort.cpp`.

```
void heapifyTernary(vector<int>& arr,int n,int i,Counter& c){
    int largest=i,l=3*i+1,m=3*i+2,r=3*i+3;
    if(l<n){ c.comparisons++; if(arr[l]>arr[largest]) largest=l; }
    if(m<n){ c.comparisons++; if(arr[m]>arr[largest]) largest=m; }
    if(r<n){ c.comparisons++; if(arr[r]>arr[largest]) largest=r; }
    if(largest!=i){ swap(arr[i],arr[largest]); c.assignments+=3;
        heapifyTernary(arr,n,largest,c); }
}
```

Listing 3: Heap Sort Ternarny: funkcja heapifyTernary

3 Porównanie Działania Algorytmów

3.1 Procedura Testowa

Dane do analizy zostały wygenerowane i zebrane w pliku `main.cpp`. Przeprowadziłem testy na losowo generowanych tablicach o trzech rozmiarach: $N = 100$, $N = 500$ i $N = 1000$. Dla każdego algorytmu i każdego rozmiaru, licznik operacji był zerowany przed sortowaniem, aby upewnić się, że dane są rzetelne.

3.2 Tabela Wyników Operacji Elementarnych

Poniższa tabela przedstawia dokładne wyniki liczby porównań i przypisań dla każdego z sześciu algorytmów. Modyfikacje zostały wyszczególnione dla każdego rozmiaru N.

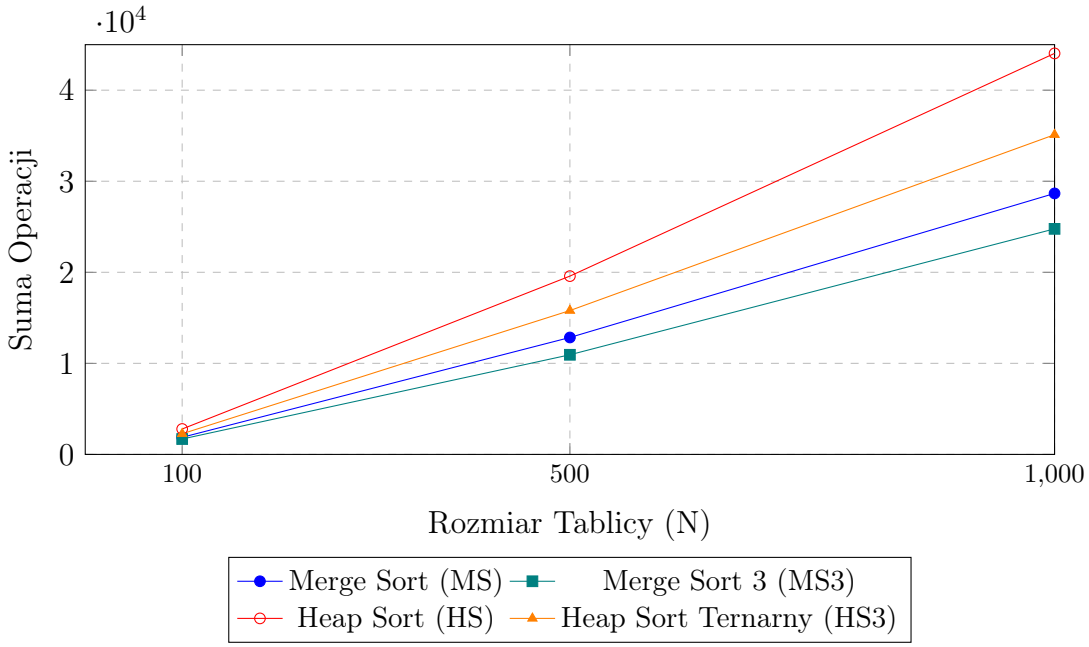
Tabela 1: Liczba Operacji Elementarnych (Porównania i Przypisania) dla Różnych Rozmiarów Tablicy

Algorytm	**N=100**		**N=500**		**N=1000**	
	Porównania	Przypisania	Porównania	Przypisania	Porównania	Przypisania
IS	2431	2536	64174	64680	266487	267493
IS Podwójny	354	525	1921	2779	4569	6330
MS	533	1344	3866	8976	8711	19952
MS 3	714	976	5042	5900	11243	13528
HS	1036	1755	7432	12150	16855	27186
HS Ternarny	1016	1251	7287	8511	16402	18711

3.3 Wykres Porównawczy Skalowania Algorytmów

Wykres przedstawia, jak całkowita liczba operacji (Suma Porównań i Przypisań) rośnie w zależności od rozmiaru tablicy N. Modyfikacje `Merge Sort` i `Heap Sort` (MS3 i HS3) wykazują wyraźnie lepsze skalowanie niż ich wersje bazowe.

Rysunek 1: Suma Operacji Elementarnych w Funkcji Rozmiaru N



4 Wnioski Końcowe

Analiza zebranych danych potwierdza, że wszystkie wprowadzone modyfikacje były opłacalne.

- Insertion Sort Double (ISD): Przyniosło największy zysk, któy wynika z drastycznego zmniejszenia liczby przypisań (przesunięć), które są głównym kosztem Insertion Sort.

- Merge Sort 3 (MS3): Lepszy niż bazowy MS. Zyskaliśmy na tym, że drzewo rekursji było płytsze ($\log_3 N$ zamiast $\log_2 N$). Mimo dodatkowych porównań, mniejsza liczba kroków rekurencyjnych zaoszczędziła więcej operacji ogółem.
- Heap Sort Ternarny (HS3): Okazał się wydajniejszy od Heap Sort (HS). Mniejsza wysokość kopca ternarnego była opłacalna i przewyższyła koszt dodatkowego porównania w procedurze `heapify`.