1. Giriş

Bu projede, iki farklı sıralama algoritması olan Merge Sort ve Quick Sort'un farklı büyüklükteki dizilerdeki performansları karşılaştırılmıştır. Deneysel sonuçlar üzerinden bu algoritmaların zaman karmaşıklıkları analiz edilmiş ve gerçek çalışma sürelerine göre c katsayıları hesaplanmıştır. Gerçek dünya koşullarında algoritmaların teorik karmaşıklıkla ne kadar uyumlu çalıştıkları araştırılmıştır.

2. Teorik Bilgi

Merge Sort algoritması her durumda Θ(n log n) zaman karmaşıklığına sahiptir. Böl ve yönet (divide and conquer) prensibiyle çalışır.

Quick Sort ise ortalama durumda Θ(n log n), en kötü durumda ise Θ(n^2) zaman karmaşıklığına sahiptir. Ancak genelde pratikte Merge Sort'tan daha hızlı çalışır.

Deneysel ölçümlerde zaman, şu formüle göre analiz edilmiştir:
$T(n) \approx c \cdot n \cdot \log_2(n)$

Buradan c katsayısı şu şekilde hesaplanmıştır:
$c = T(n) / (n \cdot \log_2(n))$

3. Kodlar

C dilinde Quick Sort ve Merge Sort algoritmaları yazılmıştır. Zaman ölçümleri clock() fonksiyonu ile yapılmıştır. Her n boyutundaki rastgele dizi için 50 farklı deneme yapılmış, ortalama süre alınmıştır.

1. Kütüphaneler ve fonksiyon imzaları

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>

// Fonksiyon prototipleri
void MergeSort(int arr[], int left, int right);
void Merge(int arr[], int left, int mid, int right);
void QuickSort(int arr[], int low, int high);
int partition(int arr[], int low, int high);
int hoare_partition(int arr[], int low, int high);
void generateRandomArray(int* arr, int size);
double measureTime(void (*sortFunction)(int*, int, int), int* arr, int size);
```

2. Mergesort ve Quicksort fonksiyonları

```
void MergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        MergeSort(arr, left, mid);
        MergeSort(arr, (mid + 1), right);

        Merge(arr, left, mid, right);
    }
}
void Merge(int arr[], int left, int mid, int right) {

    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    i = 0;
    j = 0;
    k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
void QuickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = hoare_partition(arr, low, high);

        QuickSort(arr, low, pi);
        QuickSort(arr, pi + 1, high);
    }
}

int hoare_partition(int arr[], int low, int high) {
    int pivot = arr[low];
    int i = low - 1;
    int j = high + 1;

    while (1) {
        do {
            i++;
        } while (arr[i] < pivot);

        do {
            j--;
        } while (arr[j] > pivot);

        if (i >= j)
            return j;

        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

3. Yardımcı fonksiyonlar

```
// Rastgele dizi oluşturma fonksiyonu
void generateRandomArray(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 1000000; // 0-999999 arasında rastgele sayılar
    }
}

// Zaman ölçme fonksiyonu
double measureTime(void (*sortFunction)(int*, int, int), int* arr, int
size) {
    clock_t start = clock();
    sortFunction(arr, 0, size - 1);
    clock_t end = clock();
    return (double)(end - start) / CLOCKS_PER_SEC; // saniye cinsinden
}
```

4. Main fonksiyonu

```c
int main() {

    // Rastgele sayı üreteci için başlangıç
    srand(time(NULL));

    // Sonuçları yazmak için dosya açma
    FILE* fp = fopen("sonuclar.txt", "w");
    if (fp == NULL) {
        printf("Dosya açılamadı.\n");
        return 1;
    }

    fprintf(fp, "n,merge_time,c_merge,quick_time,c_quick\n");

    // Dizi boyutları
    int sizes[6] = { 1000, 5000, 10000, 50000, 100000, 200000 };

    for (int i = 0; i < 6; i++) {
        int size = sizes[i];
        double totalMergeTime = 0;
        double totalQuickTime = 0;

        for (int j = 0; j < 50; j++) {

            // Orijinal dizi
            int* originalArray = (int*)malloc(size * sizeof(int));
            generateRandomArray(originalArray, size);

            // Kopya diziyi oluştur
            int* copy1 = (int*)malloc(size * sizeof(int));
            int* copy2 = (int*)malloc(size * sizeof(int));
            for (int k = 0; k < size; k++) {
                copy1[k] = originalArray[k];
                copy2[k] = originalArray[k];
            }

            // MergeSort ve QuickSort için zaman ölçümü
            totalMergeTime += measureTime(MergeSort, copy1, size);
            totalQuickTime += measureTime(QuickSort, copy2, size);

            // Belleği temizle
            free(originalArray);
            free(copy1);
            free(copy2);
        }

        //MergeSort için ortalama süre
        double averageMergeTime = totalMergeTime / 50;

        //QuickSort için ortalama süre
```

```
        double averageQuickTime = totalQuickTime / 50;

        // C sabitlerini hesapla
        double c_merge = averageMergeTime / (size * log2(size));
        double c_quick = averageQuickTime / (size * log2(size));

        // Sonuçları ekrana yazdır
        printf("Array size: %d\n", size);
        printf("Average Merge Sort time: %.10f saniye\n",
averageMergeTime);
        printf("C_merge: %.10f\n", c_merge);
        printf("Average Quick Sort time: %.10f saniye\n",
averageQuickTime);
        printf("C_quick: %.10f\n", c_quick);

        // Dosyaya yazdır
        fprintf(fp, "%d,%.10f,%.10f,%.10f,%.10f\n",
            size,
            averageMergeTime,
            c_merge,
            averageQuickTime,
            c_quick
        );
    }

    fclose(fp);
    printf("Sonuçlar sonuclar.txt dosyasına yazıldı.\n");
    return 0;
}
```

4. Deneysel Sonuçlar (Tablolar ve Grafikler)

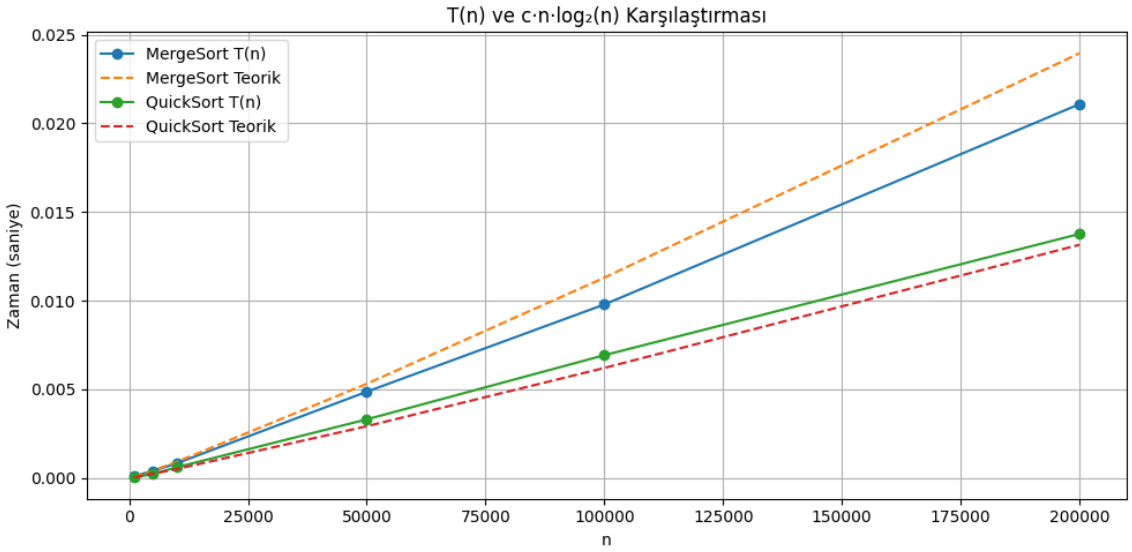
Aşağıdaki tabloda farklı dizi boyutları için T(n) ve c katsayıları verilmiştir.

c Katsayıları Tablosu

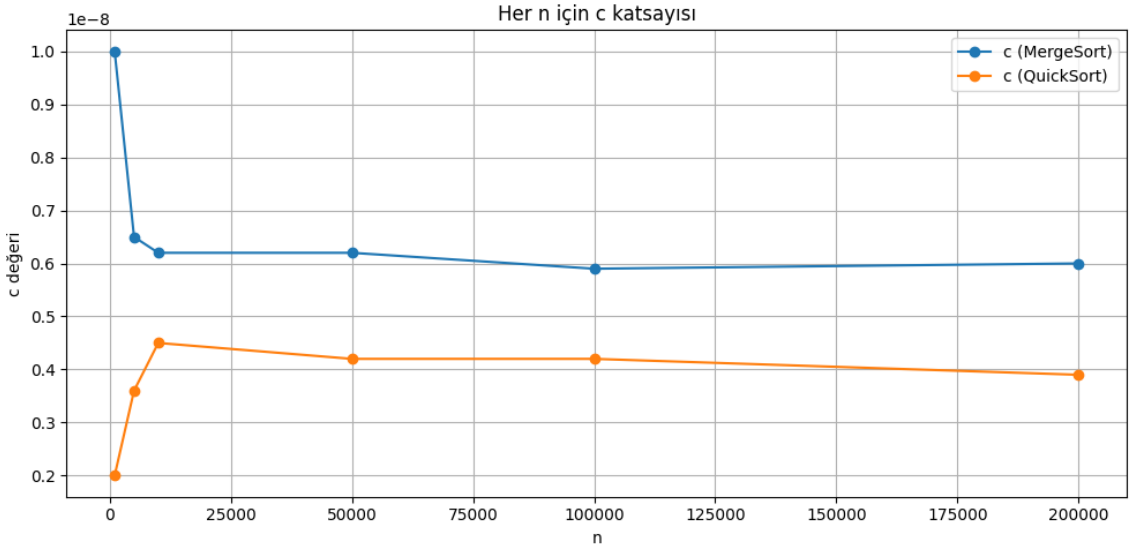
n	c_merge	c_quick
1000.0	1e-08	2e-09
5000.0	6.5e-09	3.6e-09
10000.0	6.2e-09	4.5e-09
50000.0	6.2e-09	4.2e-09
100000.0	5.9e-09	4.2e-09
200000.0	6e-09	3.9e-09

Aşağıda Merge Sort ve Quick Sort için gerçek süre ile teorik sürelerin karşılaştırmalı grafiği verilmiştir.

Grafik 1: T(n) vs c·n·log₂(n)



Grafik 2: n'e göre c katsayıları



5. Teorik ve Deneysel Kıyaslamalar

Her iki algoritma da teorik olarak Θ(n log n) karmaşıklığa sahiptir. Deneysel sonuçlar da bu karmaşıklığı genel olarak doğrulamaktadır.

Quick Sort'un c değeri Merge Sort'a kıyasla genellikle daha düşük çıkmıştır, bu da pratikte daha hızlı çalıştığını göstermektedir.

Küçük n değerlerinde c katsayısı daha dengesizdir. Bunun sebebi sistem kaynakları, cache etkileri, recursive çağrı maliyetleri ve zaman ölçüm hassasiyeti olabilir.

6. Sonuç

Sonuç olarak, hem Merge Sort hem Quick Sort teorik karmaşıklıklarını büyük oranda sağlamışlardır. Ancak Quick Sort pratikte daha hızlı sonuçlar vermiştir.

Deneysel c değerleri algoritmaların sabit zaman maliyetlerini de ortaya koymaktadır. Özellikle sistem kaynakları, CPU cache, branch prediction gibi donanım detayları da algoritma performansını etkileyebilir.

Büyük veri boyutlarında Merge Sort (her durumda nlogn) daha stabil bir davranış gösterirken, Quick Sort genellikle daha hızlıdır.

Bu çalışmada kullanılan tüm kaynak kodlara ve ek dosyalara aşağıdaki GitHub deposu üzerinden erişilebilir:

https://github.com/melihboyaci/QuickMergeSortAnalysis