Homework 7: Performance Measure and Discussion

시스템 프로그래밍 1분반 소프트웨어학과 32220236 권대근

1. Introduction

1.1 Motivation

프로그램에서 정렬은 매우 자주 사용되는 기본 연산이다.
예를 들어 데이터베이스 검색, 로그 정렬, 점수 순위를 매기는 작업 등 여러 상황에서 정렬 알고리즘의 성능은 전체 프로그램 실행 시간에 큰 영향을 준다.

이 과제에서는 서로 다른 정렬 알고리즘들의 성능 차이를 실제로 측정해 보고,
수업 시간에 배운 시간 복잡도(Big-O)와 실제 실행 시간이 어떻게 연결되는지 확인하는 것을 목표로 한다.

1.2 Objective

다음 세 가지 정렬 알고리즘의 실행 시간을 비교한다.

- **Bubble Sort**: 단순한 $O(N^2)$ 정렬 알고리즘
- **Merge Sort**: 분할 정복 기반 O(N log N) 정렬 알고리즘
- **Quick Sort**: 평균 O(N log N) 정렬 알고리즘, 피벗 기반 분할 정복

배열 크기 N을 점점 키워 가면서, 세 알고리즘의 **실행 시간(ms)**을 측정하고 비교한다.

1.3 Method

- **언어**: JavaScript
- **실행 환경**: 웹 브라우저
  - 크롬(Chrome)과 사파리(Safari) 두 브라우저에서 동일한 코드를 실행하여 결과를 비교
- **시간 측정 도구**:
  - 과제에서 요구한 `gettimeofday()` 또는 기타 도구 중,
    브라우저에서 제공하는 **`performance.now()`**를 사용
- **측정 방법 개요**:
  1. 각 크기 N에 대해 **랜덤 배열 하나 생성**
  2. 이 배열을 복사해 `bubbleSort`, `mergeSort`, `quickSort` 각각의 입력으로 사용
     (모든 알고리즘이 **완전히 동일한 데이터**를 정렬하도록 함)
  3. 각 알고리즘을 여러 번(기본 5회) 실행하여 **평균 실행 시간**을 사용
  4. Chart.js를 이용해 **배열 크기 vs 실행 시간**을 선 그래프로 시각화
  5. 같은 실험을 **크롬과 사파리에서 각각 수행**하여, 브라우저 간 차이도 함께 관찰

---

2. Test Program / Source Code

2.1 실행 환경

- OS: (예시) macOS 10.15.x
- Browsers:

    - Google Chrome 142.x
    - Safari (실험 시점 최신 버전)
- CPU: (예시) Intel Core i5 / 8GB RAM
- Tools:
    - JavaScript
    - `performance.now()` (고해상도 타이머)
    - Chart.js (그래프 출력 라이브러리)

2.2 정렬 알고리즘 소스 코드

2.2.1 Bubble Sort

```
function bubbleSort(arr) {
    const a = arr.slice();
    const arrLength = a.length;
    for (let i = 0; i < arrLength - 1; i++){
        for (let j = 0; j < arrLength - 1 - i; j++){
            if (a[j] > a[j+1]){
                const tmp = a[j];
                a[j] = a[j+1];
                a[j+1] = tmp;
            }
        }
    }
    return a;
}
```

2.2.2 Merge Sort

```
function mergeSort(arr){
    if (arr.length <= 1) return arr.slice();

    const mid = arr.length >> 1;
    const left = mergeSort(arr.slice(0, mid));
    const right = mergeSort(arr.slice(mid));

    return merge(left, right);
}

function merge(left, right){
    const result = [];
    let i = 0;
    let j = 0;

    while (i < left.length && j < right.length){
        if (left[i] <= right[j]){
            result.push(left[i++]);
        }
        else{
```

```
            result.push(right[j++]);
        }
    }
    while (i < left.length){
        result.push(left[i++]);
    }
    while (j < right.length){
        result.push(right[j++]);
    }
    return result;
}
```

2.2.3 Quick Sort

```
function quickSort(arr){
    if (arr.length <= 1){
        return arr.slice();
    }

    const a = arr.slice();
    const pivot = a[a.length >> 1];
    const left = [];
    const mid = [];
    const right = [];

    for (const x of a){
        if (x < pivot){
            left.push(x);
        }
        else if (x > pivot){
            right.push(x);
        }
        else{
            mid.push(x);
        }
    }

    return quickSort(left).concat(mid, quickSort(right));
}
```

2.3 데이터 생성 및 시간 측정 코드

```
function makeRandomArray(n){
    const arr = new Array(n);
    for (let i = 0; i < n; i++){
        arr[i] = Math.floor(Math.random() * 1_000_000);
    }
    return arr;
}
```

```
function measureSort(sortFn, baseArr, repeat = 5){
    let sum = 0;
    for (let r = 0; r < repeat; r++){
        const arr = baseArr.slice();
        const timeStart = performance.now();
        const sorted = sortFn(arr);
        const timeEnd = performance.now();

        for (let i = 1; i < sorted.length; i++){
            if (sorted[i-1] > sorted[i]){
                console.error("정렬 실패 인덱스", i);
                break;
            }
        }
        sum += (timeEnd - timeStart);
    }
    return sum / repeat;
}
```

2.4 전체 실험 및 그래프 코드

```
async function runExperiment(){
    const sizes = [1000, 4000, 7000, 10000, 13000];

    const bubbleTimes = [];
    const mergeTimes = [];
    const quickTimes = [];

    for (const n of sizes){
        console.log(`N = ${n} 테스트`);
        const baseArr = makeRandomArray(n);

        const bubbleTime = measureSort(bubbleSort, baseArr);
        const mergeTime = measureSort(mergeSort, baseArr);
        const quickTime = measureSort(quickSort, baseArr);

        bubbleTimes.push(bubbleTime);
        mergeTimes.push(mergeTime);
        quickTimes.push(quickTime);

        console.log(
          `N=${n}, bubble=${bubbleTime.toFixed(3)}ms, ` +
          `merge=${mergeTime.toFixed(3)}ms,
quick=${quickTime.toFixed(3)}ms`
        );
    }

    const ctx = document.getElementById("sortChart").getContext("2d");

    new Chart(ctx,{
```

```
        type: "line",
        data: {
            labels: sizes,
            datasets: [
                {
                    label: "Bubble Sort",
                    data: bubbleTimes,
                    borderWidth: 2,
                },
                {
                    label: "Merge Sort",
                    data: mergeTimes,
                    borderWidth: 2,
                },
                {
                    label: "Quick Sort",
                    data: quickTimes,
                    borderWidth: 2,
                },
            ],
        },
        options: {
            responsive: true,
            plugins: {
                title: {
                    display: true,
                    text: "정렬 알고리즘 별 실행 시간 (ms)",
                },
            },
            scales: {
                x: {
                    title: {
                        display: true,
                        text: "배열 크기 N",
                    },
                },
                y: {
                    title: {
                        display: true,
                        text: "평균 실행 시간 (ms)"
                    }
                }
            }
        }
    })
}

runExperiment();
```

---

3. Snapshots

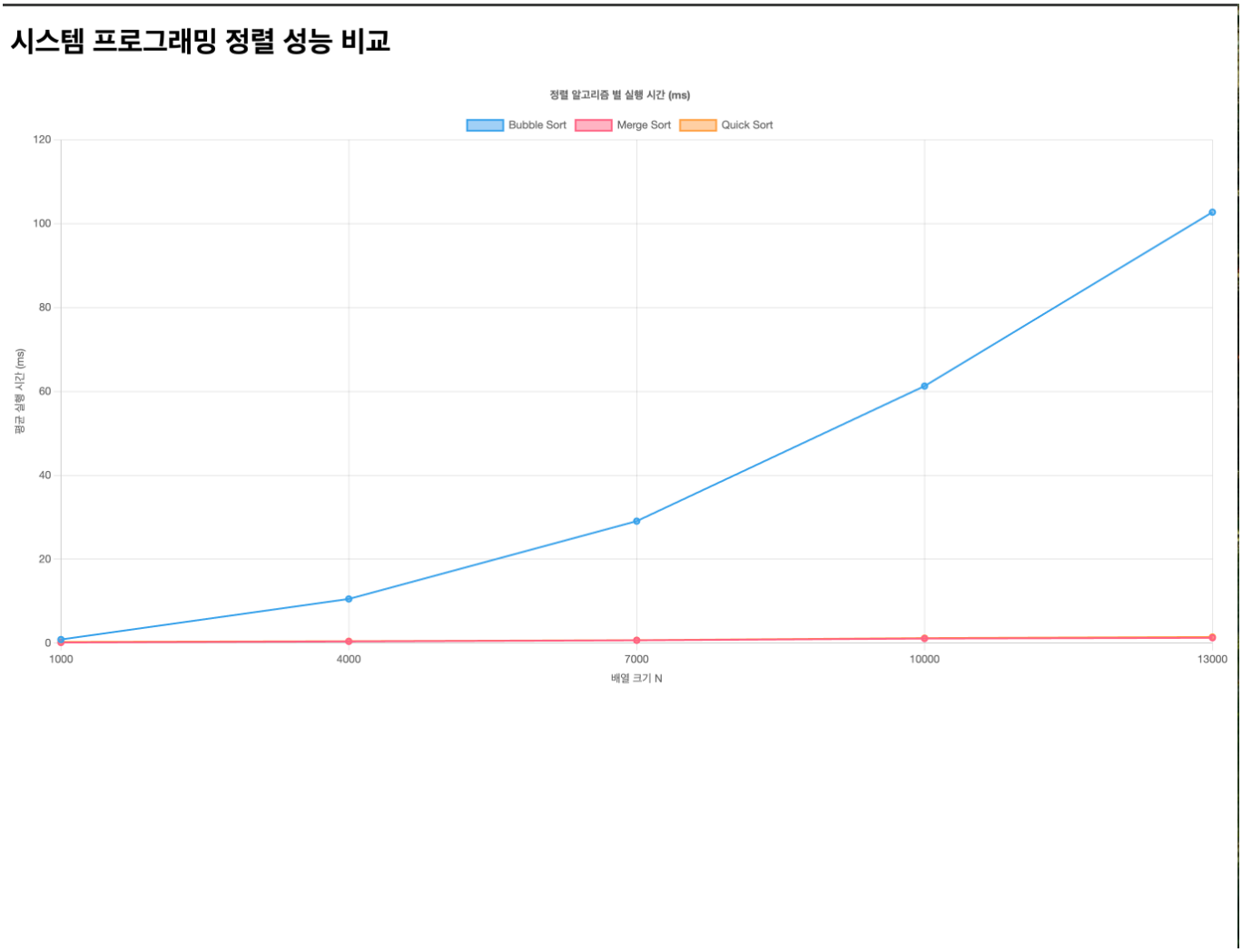
3.1 그래프 스냅샷 (Chrome)

- **그림 1. 크롬 브라우저에서 정렬 알고리즘 별 실행 시간 그래프**

> 그림 1은 배열 크기 N(1000, 4000, 7000, 10000, 13000)에 대해
> 크롬 브라우저에서 측정한 Bubble Sort, Merge Sort, Quick Sort의
> 평균 실행 시간을 선 그래프로 나타낸 것이다.
> N이 커질수록 Bubble Sort의 실행 시간이 빠르게 증가하는 반면,
> Merge Sort와 Quick Sort는 비교적 완만한 증가를 보인다.

3.2 실행 결과 및 그래프 이미지 (Chrome)

```
N = 1000 테스트                                                    script.js:109
N=1000, bubble=0.860ms, merge=0.140ms, quick=0.280ms               script.js:120
N = 4000 테스트                                                    script.js:109
N=4000, bubble=10.520ms, merge=0.400ms, quick=0.460ms              script.js:120
N = 7000 테스트                                                    script.js:109
N=7000, bubble=29.080ms, merge=0.660ms, quick=0.700ms              script.js:120
N = 10000 테스트                                                   script.js:109
N=10000, bubble=61.280ms, merge=1.080ms, quick=1.200ms             script.js:120
N = 13000 테스트                                                   script.js:109
N=13000, bubble=102.740ms, merge=1.260ms, quick=1.460ms            script.js:120
>
```



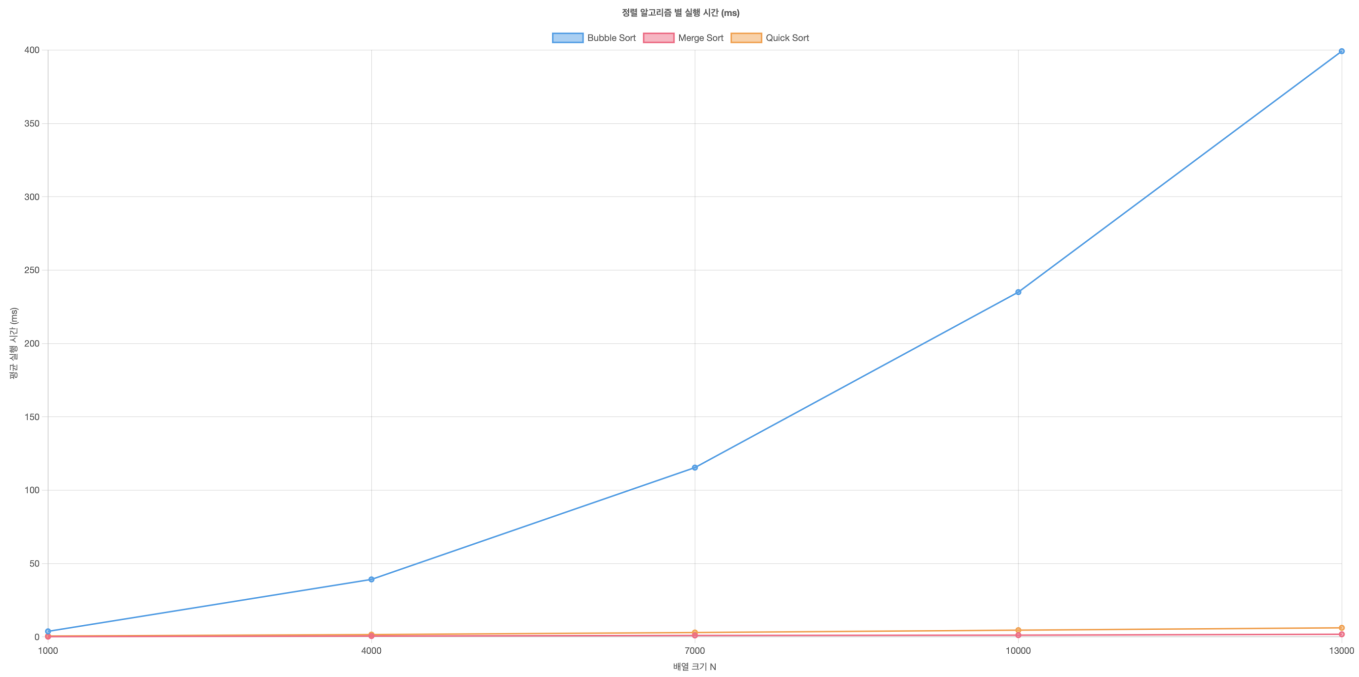
3.3 그래프 스냅샷 (Safari)

- **그림 2. 사파리 브라우저에서 정렬 알고리즘 별 실행 시간 그래프**

> 그림 2는 동일한 배열 크기 N(1000, 4000, 7000, 10000, 13000)에 대해
> 사파리 브라우저에서 측정한 결과를 나타낸 그래프이다.
> 절대적인 실행 시간에는 크롬과 약간의 차이가 있었지만,
> 세 알고리즘의 상대적인 순서와 증가하는 패턴은 거의 동일하게 나타났다.

```
N = 1000 테스트
N=1000, bubble=3.800ms, merge=0.200ms, quick=0.600ms
N = 4000 테스트
N=4000, bubble=39.200ms, merge=0.600ms, quick=1.600ms
N = 7000 테스트
N=7000, bubble=115.400ms, merge=1.000ms, quick=3.000ms
N = 10000 테스트
N=10000, bubble=235.000ms, merge=1.200ms, quick=4.600ms
N = 13000 테스트
N=13000, bubble=399.200ms, merge=1.800ms, quick=6.200ms
```

시스템 프로그래밍 정렬 성능 비교



3.4 측정 데이터 테이블

3.4.1 Chrome

N	Bubble Sort (ms)	Merge Sort (ms)	Quick Sort (ms)
1000	0.860	0.140	0.280

N	Bubble Sort (ms)	Merge Sort (ms)	Quick Sort (ms)
4000	10.520	0.400	0.460
7000	29.080	0.660	0.700
10000	61.280	1.080	1.200
13000	102.740	1.260	1.460

3.4.2 Safari

N	Bubble Sort (ms)	Merge Sort (ms)	Quick Sort (ms)
1000	3.800	0.200	0.600
4000	39.200	0.600	1.600
7000	115.400	1.000	3.000
10000	235.000	1.200	4.600
13000	399.200	1.800	6.200

---

4. Discussion

4.1 Bubble Sort의 성능 특성

Bubble Sort는 두 개의 중첩 반복문을 사용하는 단순한 알고리즘으로,
이론적인 시간 복잡도는 **O(N²)**이다.

실험 결과에서도 이 특징이 뚜렷하게 나타났다.
크롬 환경에서 Bubble Sort의 실행 시간은 다음과 같이 증가했다.

- N = 1000 → **0.860 ms**
- N = 4000 → **10.520 ms**
- N = 7000 → **29.080 ms**
- N = 10000 → **61.280 ms**
- N = 13000 → **102.740 ms**

입력 크기 N이 약 13배(1000 → 13000)로 증가하는 동안, 실행 시간은 0.86 ms에서 약 102.74 ms로 크게 증가하였다.
정확히 N² 비율로 증가하는 것은 아니지만, **선형보다 훨씬 빠른 속도**로 증가하는 것을 통해
O(N²) 알고리즘의 비효율성을 확인할 수 있다.

사파리에서도 비슷한 패턴을 보였다.

- N = 1000 → **3.800 ms**
- N = 4000 → **39.200 ms**
- N = 7000 → **115.400 ms**
- N = 10000 → **235.000 ms**
- N = 13000 → **399.200 ms**

특히 N = 13000에서 사파리의 Bubble Sort 시간은 약 399.2 ms로,
같은 조건의 크롬(102.74 ms)보다 약 4배 느리게 측정되었다.
하지만 **두 브라우저 모두에서 “N이 커질수록 Bubble Sort가 폭발적으로 느려진다”**는 점은 동일하다.
이 결과는 이론적인 O(N²) 복잡도와 잘 맞는 경향을 보여 준다.

4.2 Merge Sort와 Quick Sort의 성능 비교

Merge Sort와 Quick Sort는 이론적으로 **평균 O(N log N)** 시간 복잡도를 가진다.
실험에서 측정된 실행 시간은 Bubble Sort보다 훨씬 작게 나타났다.

크롬 기준:

- N = 13000일 때
  - Bubble: **102.740 ms**
  - Merge: **1.260 ms**
  - Quick: **1.460 ms**

사파리 기준:

- N = 13000일 때
  - Bubble: **399.200 ms**
  - Merge: **1.800 ms**
  - Quick: **6.200 ms**

입력 크기가 커져도 Merge Sort와 Quick Sort는
거의 **1~6 ms 수준**에서 머무르는 반면,
Bubble Sort는 **수십~수백 ms**까지 치솟는다.

또한 두 브라우저 모두에서 **Merge Sort와 Quick Sort의 실행 시간 차이는 상대적으로 작고,**
항상 Bubble Sort보다 훨씬 빠르다는 점도 공통적으로 관찰되었다.
이것은 O(N log N) 알고리즘이 O(N²) 알고리즘에 비해
큰 입력에서 얼마나 유리한지 잘 보여 준다.

4.3 크롬과 사파리 브라우저 성능 비교

같은 JavaScript 코드와 같은 데이터 크기를 사용했지만,
크롬과 사파리에서 측정된 절대 실행 시간에는 차이가 있었다.

- Bubble Sort의 경우, 사파리에서 항상 크롬보다 느리게 측정되었다.
  - 예: N = 13000 기준
    - 크롬: **102.740 ms**
    - 사파리: **399.200 ms** (약 3.9배 느림)
- Merge Sort와 Quick Sort도 사파리 쪽이 약간 느리게 나왔지만,
  그 차이는 Bubble Sort만큼 극적이진 않았다.
  - 예: N = 13000 기준
    - Merge: 크롬 1.260 ms vs 사파리 1.800 ms
    - Quick: 크롬 1.460 ms vs 사파리 6.200 ms

이 차이는 두 브라우저가 서로 다른 JavaScript 엔진을 사용하기 때문이라고 볼 수 있다.

- 크롬: **V8 엔진**
- 사파리: **JavaScriptCore 엔진**

엔진 내부의 최적화 전략, JIT 컴파일 방식, 메모리 관리 등의 구현 차이 때문에
같은 자바스크립트 코드라도 브라우저마다 **상수 시간 계수**가 달라질 수 있다.

하지만 중요한 점은, **브라우저가 달라져도 알고리즘 간 상대적인 순서와 패턴은 동일**했다는 것이다.

- 두 브라우저 모두에서 Bubble Sort가 가장 느리고,
  Merge Sort와 Quick Sort는 매우 빠르게 동작했다.
- N을 키워도 Merge/Quick의 증가 폭은 완만하고,
  Bubble은 급격히 증가하는 모양이 그대로 유지되었다.

즉, 실행 환경에 따라 절대적인 속도는 달라질 수 있지만,
알고리즘의 시간 복잡도 차이가 성능에 미치는 영향이 훨씬 더 크다는 것을 확인할 수 있었다.

4.4 알고리즘 선택의 중요성

이번 실험에서 사용한 최대 입력 크기 N은 13,000으로,
실제 산업 현장에서 다루는 수십만~수백만 개의 데이터에 비하면 작은 편이다.
그럼에도 불구하고, **Bubble Sort와 O(N log N) 알고리즘(Quick / Merge) 사이의 성능 차이는 이미 매우 크게 나타났다.**

- 크롬에서 N = 13000일 때
  - Bubble: ~102.7 ms
  - Merge/Quick: ~1.2~1.4 ms 수준
- 사파리에서 N = 13000일 때
  - Bubble: ~399.2 ms
  - Merge/Quick: ~1.8~6.2 ms 수준

입력 크기가 더 커질수록 이 격차는 기하급수적으로 벌어질 것이다.
따라서 실제 서비스나 프로그램을 설계할 때는
구현이 단순하다는 이유만으로 $O(N^2)$ 알고리즘을 선택하는 것은 위험하며,
가능하다면 O(N log N) 또는 그보다 더 효율적인 알고리즘을 선택해야 한다.

4.5 측정 방식 및 한계

- 본 실험은 브라우저에서 JavaScript와 `performance.now()`를 사용하여 측정하였다.
  이는 과제 요구 사항의 “gettimeofday() or other tools” 중 **other tools**에 해당한다.
- 브라우저 환경에서는 백그라운드 탭, 다른 프로그램 실행, OS 스케줄링 등으로 인해
  실행 시간에 **노이즈**가 포함될 수 있다.
- 같은 코드를 다른 PC나 다른 OS에서 실행하면 절대적인 ms 값은 달라질 수 있다.
  그러나 이번 실험에서 보았듯이,
  알고리즘 간 상대적인 증가 추세와 순서는 두 브라우저에서 모두 일관되게 유지되었다.
- 또한 실험에서는 **랜덤 배열**만 사용했지만,
  이미 정렬된 배열이나 역정렬 배열과 같이 입력 분포가 크게 다른 경우에는
  특히 Quick Sort의 성능이 영향을 받을 수 있다.
  이 부분은 추가 실험 주제로 남겨 둘 수 있다.

4.6 배운 점

이번 과제를 통해 다음과 같은 점을 배웠다.

1. 단순히 "코드가 정상적으로 동작한다"에서 끝나는 것이 아니라,
   입력 크기가 커졌을 때 성능이 어떻게 변하는지 수치로 확인하는 작업이 중요하다는 것을 느꼈다.
2. 이론에서 배운 시간 복잡도
   - Bubble Sort: $O(N^2)$
   - Merge / Quick Sort: $O(N \log N)$
     이 실제 실행 시간의 증가 패턴과 매우 잘 맞는다는 사실을 확인할 수 있었다.
3. 같은 알고리즘이라도 브라우저(크롬, 사파리)와 JavaScript 엔진에 따라
   절대 실행 시간은 달라질 수 있지만,
   알고리즘 선택이 성능에 미치는 영향이 엔진 차이보다 훨씬 크다는 점을 실감했다.
4. 앞으로 실제 프로젝트나 과제에서 알고리즘을 선택할 때,
   구현 난이도뿐 아니라 **시간 복잡도, 데이터 크기, 실행 환경(브라우저, 언어, 런타임 등)**을
   함께 고려해야 한다는 점을 다시 한 번 깨닫게 되었다.