Sorting Algorithm

2021학년도 1학기 컴퓨터알고리즘 9주차 과제

> 이름: 최종민
>
> 학번: 202001677

Sorting Algorithm(정렬 알고리즘)

정렬 알고리즘은 크게 `내부정렬(Internal Sort)`과 `외부정렬(External Sort)`로 분류한다.

`내부정렬`은 입력의 크기가 주기억 장치(main memory)의 공간보다 크지 않은 경우에 수행되는 정렬이다.

그러나 입력의 크기가 주기억 장치 공간보다 큰 경우에는, 보조 기억 장치에 있는 입력을 여러 번에 나누어 주기억 장치에 읽어 들인 후, 정렬하여 보조 기억 장치에 다시 저장하는 과정을 반복해야 한다. 이러한 정렬을 `외부정렬`이라고 한다.

이 문서에는 내부정렬 중에서 `버블 정렬(Bubble Sort)`, `선택 정렬(Selection Sort)`, `삽입 정렬(Insertion Sort)`, `쉘 정렬(Shell Sort)`에 대해 다룬다.

Bubble Sort(버블 정렬)

`버블 정렬`은 이웃하는 숫자를 비교하여 작은 수를 앞쪽으로 이동시키는 과정을 반복하여 정렬하는 알고리즘이다.

BubbleSort.java

```java
public class BubbleSort extends Sort {

    @Override
    public int[] sort(int[] array) {
        System.out.println("\n=== Bubble Sort ===");
        int length = array.length;
        int[] result = array.clone();

        long startTime = System.currentTimeMillis();
        for (int i = 1; i <= length - 1; i++)
            for (int j = 1; j <= length - i; j++)
                if (result[j - 1] > result[j]) {
                    int temp = result[j - 1];
                    result[j - 1] = result[j];
                    result[j] = temp;
                }
        long endTime = System.currentTimeMillis();

        measureTime(startTime, endTime);
        return result;
    }
}
```

첫 번째 for 루프가 배열의 길이인 length -1번 반복되고, 두 번째 for 루프는 j가 length - i 번 반복되는데, 앞 부분은 이미 정렬된 상태이므로 비교할 필요가 없기 때문이다.

만약 result[j - 1] 이 result[j] 보다 클 경우, 이 둘의 값을 바꿔준다.

for 루프가 모두 끝나면 정렬이 끝났으므로, result 를 리턴한다.

Selection Sort(선택 정렬)

선택 정렬 은 배열 전체에서 최솟값을 선택하여 i 번째 원소와 자리를 바꾼다.

이 과정을 반복하다 마지막에 2개의 원소 중에서 작은 값을 선택해 자리를 바꿈으로써 정렬하는 알고리즘이다.

SelectionSort.java

```java
public class SelectionSort extends Sort {

    @Override
    public int[] sort(int[] array) {
        System.out.println("\n=== Selection Sort ===");
        int length = array.length;
        int[] result = array.clone();

        long startTime = System.currentTimeMillis();
        for (int i = 0; i <= length - 2; i++) {
            int min = i;
            for (int j = i + 1; j <= length - 1; j++)
                if (result[j] < result[min])
                    min = j;
            int temp = result[i];
            result[i] = result[min];
            result[min] = temp;
        }
        long endTime = System.currentTimeMillis();

        measureTime(startTime, endTime);
        return result;
    }
}
```

첫 번째 for 루프는 length -1번 반복하며, 먼저 i 를 가장 작은 원소의 인덱스로 설정하여 min 에 저장한다.

두 번째 for 루프에서는 j 를 i +1로 설정하여 i 오른쪽의 원소부터 min 을 인덱스로 갖는 원소와 비교한다.

이 과정에서 더 작은 값을 갖는 인덱스를 다시 min 으로 지정한다.

두 번째 for 루프를 벗어나면, i 번째 원소와 min 번째 원소를 서로 교환한다.

for 루프가 모두 끝나면 정렬이 끝났으므로, result 를 리턴한다.

Insertion Sort(삽입 정렬)

삽입 정렬은 배열을 정렬된 부분(앞)과 되지 않은 부분(뒤)으로 나누고, 안 된 부분의 가장 왼쪽 원소를 정렬된 부분에 삽입하는 과정을 반복하여 정렬하는 알고리즘이다.

InsertionSort.java

```java
public class InsertionSort extends Sort {

    @Override
    public int[] sort(int[] array) {
        System.out.println("\n=== Insertion Sort ===");
        int length = array.length;
        int[] result = array.clone();

        long startTime = System.currentTimeMillis();

        for (int i = 1; i ⩽ length - 1; i++) {
            int CurrentElement = result[i];
            int j = i - 1;
            while (j ⩾ 0 && result[j] > CurrentElement) {
                result[j + 1] = result[j];
                j--;
            }
            result[j + 1] = CurrentElement;
        }

        long endTime = System.currentTimeMillis();

        measureTime(startTime, endTime);
        return result;
    }
}
```

for 루프를 length -1번 반복하며, CurrentElement 는 정렬이 안 된 부분의 가장 왼쪽 원소를 나타낸다.

j 를 정렬된 부분의 가장 오른쪽 원소부터 감소하면서 원소에 접근하기 위해 i -1로 초기값을 설정한다.

j 가 배열의 범위를 벗어나지 않게 하고, result[j] 가 CurrentElement 보다 큰 동안 while 루프를 수행한다.

while 루프 안에서는 원소를 오른쪽으로 자리를 이동한다.

while 루프가 끝나면, 다시 for 문을 수행해야 하므로, result[j] 에 CurrentElement 를 삽입한다.

반복문이 모두 끝나면 정렬이 끝났으므로, result 를 리턴한다.

Shell Sort(쉘 정렬)

쉘 정렬은 삽입 정렬을 이용해 배열 뒷 부분의 작은 숫자를 빠르게 앞 부분으로, 큰 숫자를 뒷 부분으로 이동시켜 마지막에 삽입 정렬을 수행하여 정렬하는 알고리즘이다.

```java
public class ShellSort extends Sort {

    @Override
```

```
    public int[] sort(int[] array) {
        System.out.println("\n=== Shell Sort ===");
        int length = array.length;
        int[] result = array.clone();

        long startTime = System.currentTimeMillis();

        for (int h = length / 3; h > 0; h = h / 3)
            for (int i = h; i ⩽ length - 1; i++) {
                int CurrentElement = result[i];
                int j = i;
                while (j ⩾ h && result[j - h] > CurrentElement) {
                    result[j] = result[j - h];
                    j = j - h;
                }
                result[j] = CurrentElement;
            }

        long endTime = System.currentTimeMillis();

        measureTime(startTime, endTime);
        return result;
    }
}
```

먼저, 여러 인덱스씩 이동하기 위한 간격 h를 length / 3으로 설정한다.

h가 0보다 클 때까지 첫 번째 for루프를 수행하고, i를 h부터 length -1까지 수행시킨다.

CurrentElement를 result[i]로 설정하고, j를 i부터 while루프를 수행한다.

이 while루프는 result[j - h]가 CurrentElement보다 큰 경우, 원소를 뒤로 이동시키는 역할을 한다.

반복문이 모두 끝나면 정렬이 끝났으므로, result를 리턴한다.

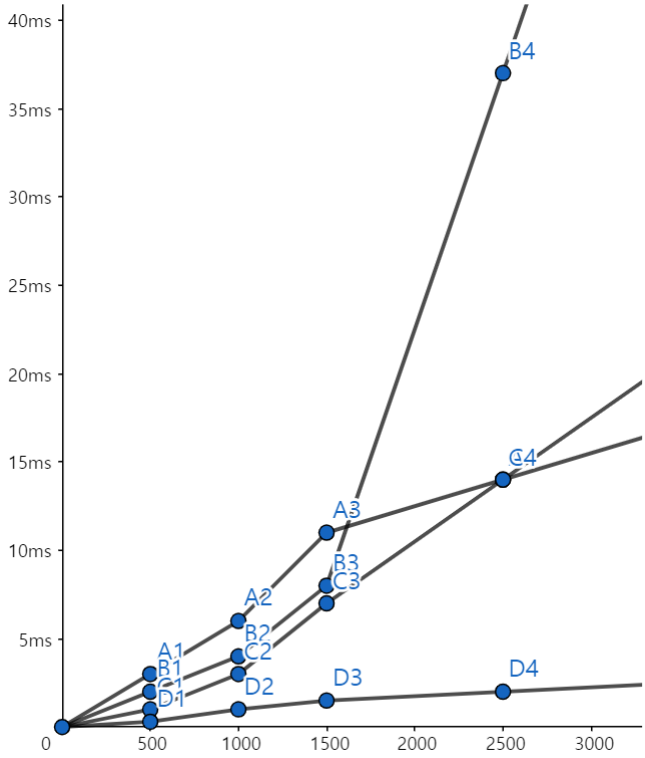
성능 분석

버블 정렬(Bubble Sort)의 시간복잡도는
$n(n-1)/2 \times O(1) = (1/2n^2 - 1/2n) \times O(1) = O(n^2) \times O(1) = O(n^2)$ 이다.

선택 정렬(Selection Sort)의 시간복잡도는 $n(n-1)/2 \times O(1) = O(n^2)$ 이다. 특히 항상 일정한 시간복잡도를 나타내는 것이 특징이며, 입력에 민감하지 않은(input insensitive) 알고리즘이다.

삽입 정렬(Insertion Sort)의 시간복잡도는 $n(n-1)/2 \times O(1) = O(n^2)$이다. 이 정렬은 입력에 따라 수행 시간이 달라질 수 있다.

쉘 정렬(Shell Sort)의 시간복잡도는 최악의 경우 $O(n^2)$, 히바드(Hibbard)의 간격($2^k - 1$)을 사용하면 $O(n^{1.5})$가 된다. 실험을 통해 $O(n^{1.25})$가 됨을 알아냈지만, 간격에 따라 시간복잡도가 달라져 풀리지 않은 문제로 남아있다.



예를 들어, 같은 랜덤 배열로 위 네 가지 알고리즘을 모두 실행한 결과, A, B, C는 각각 버블 정렬, 선택 정렬, 삽입 정렬을 나타내는데, 이론과 시간복잡도가 유사한 것을 볼 수 있다.

D는 쉘 정렬로, 이 또한 이론과 유사한 것을 확인할 수 있다.

참고 및 출처

- 알기 쉬운 알고리즘(양성봉)