

알고리즘 설계와 분석 Project1



서강대학교
SOGANG UNIVERSITY

담당 교수 : 임인성

이름 : 박민준

학번 : 20212020

1. Algorithm Description

1.1. Algorithm 3: 시간 복잡도 $O(n^4)$

```
// Algorithm 3:  $O(n^4)$  Summed Area Table
int MaxSumSubrectangle3(vector<vector<int>>& mat, int& k, int& i, int& l, int& j) {
    int n = mat.size();
    int max_sum = INT_MIN;
    vector<vector<int>> sum(n + 1, vector<int>(n + 1, 0));

    for (int r = 1; r <= n; r++) {
        for (int c = 1; c <= n; c++) {
            sum[r][c] = mat[r - 1][c - 1] + sum[r - 1][c] + sum[r][c - 1] - sum[r - 1][c - 1];
        }
    }

    for (int r1 = 0; r1 < n; r1++) {
        for (int c1 = 0; c1 < n; c1++) {
            for (int r2 = r1; r2 < n; r2++) {
                for (int c2 = c1; c2 < n; c2++) {
                    int current_sum = sum[r2 + 1][c2 + 1] - sum[r1][c2 + 1] - sum[r2 + 1][c1] + sum[r1][c1];
                    if (current_sum > max_sum) {
                        max_sum = current_sum;
                        k = r1;
                        i = c1;
                        l = r2;
                        j = c2;
                    }
                    else if (current_sum == max_sum) {
                        if (c1 < i || (c1 == i && c2 < j) || (c1 == i && c2 == j && r1 < k) || (c1 == i && c2 == j && r1 == k && r2 < l)) {
                            k = r1;
                            i = c1;
                            l = r2;
                            j = c2;
                        }
                    }
                }
            }
        }
    }

    return max_sum;
}
```

- 해당 알고리즘은 Summed Area Table을 사용하여 2D 행렬에서 모든 가능한 직사각형의 합을 계산하는 방식이다.
- 먼저 행렬의 사이즈 n 과 maximum sum값을 계산할 `max_sum` 변수, 그리고 Summed Area Table에 해당하는 2차원 배열 `sum`을 선언해준 뒤 이중 for문을 통해 Summed Area Table을 세팅한다.
- Summed Area Table은 각 좌표까지의 누적 합을 미리 계산하여 특정 부분 직사각형의 합을 빠르게 구할 수 있게 해준다.
- Summed Area Table을 세팅한 뒤, 4번의 중첩 for문을 사용해 행렬 내의 모든 직사각형을 순회하며 최대 구간 합을 가지는 직사각형의 좌측 상단 꼭짓점의 좌표, 우측 하단 꼭짓점의 좌표를 반환한다.
- 최대 합을 찾을 때, $i \rightarrow j \rightarrow k \rightarrow l$ 순서로 인덱스가 작은 subrectangle을 선택한다.

1.2. Algorithm4: 시간 복잡도 $O(n^3 \log n)$

```
// Algorithm 4:  $O(n^3 \log n)$  - 열을 합친 후 분할 정복 기법을 적용
int MaxSumSubrectangle4(vector<vector<int>>& mat, int& k, int& i, int& l, int& j) {
    int n = mat.size();
    int max_sum = INT_MIN;

    for (int left = 0; left < n; left++) {
        vector<int> tmp(n, 0);
        for (int right = left; right < n; right++) {
            for (int row = 0; row < n; row++)
                tmp[row] += mat[row][right];

            int start, end;
            int sum = max_subarraySum(tmp, 0, n - 1, start, end);

            // 최대 합을 갱신하고, 직사각형 좌표 저장
            if (sum > max_sum) {
                max_sum = sum;
                k = start; // 시작 행 (최대 합이 발생한 구간의 시작 행)
                i = left;  // 시작 열
                l = end;   // 끝 행 (최대 합이 발생한 구간의 끝 행)
                j = right; // 끝 열
            }
        }
    }

    return max_sum;
}
```

- 해당 알고리즘은 먼저 각 열을 합산하여 2D 배열을 1D 배열로 변환한 다음, 1D 배열에서 분할 정복 기법(Divide-And-Conquer strategy)을 사용해 최대 구간 합을 찾는다.

```
int max_subarraySum(vector<int>& arr, int left, int right, int& start, int& end) {
    if (left == right) {
        start = end = left;
        return arr[left];
    }

    int mid = (left + right) / 2;
    int left_start, left_end, right_start, right_end, cross_start, cross_end;

    int left_max = max_subarraySum(arr, left, mid, left_start, left_end);
    int right_max = max_subarraySum(arr, mid + 1, right, right_start, right_end);
    int cross_max = max_crossSum(arr, left, mid, right, cross_start, cross_end);

    if (left_max >= right_max && left_max >= cross_max) {
        start = left_start;
        end = left_end;
        return left_max;
    }
    else if (right_max >= left_max && right_max >= cross_max) {
        start = right_start;
        end = right_end;
        return right_max;
    }
    else {
        start = cross_start;
        end = cross_end;
        return cross_max;
    }
}
```

```

int max_crossSum(vector<int>& arr, int left, int mid, int right, int& start, int& end) {
    int sum = 0;
    int left_sum = INT_MIN;
    int tmp_start = mid;
    for (int i = mid; i >= left; i--) {
        sum += arr[i];
        if (sum > left_sum) {
            left_sum = sum;
            tmp_start = i;
        }
    }

    sum = 0;
    int right_sum = INT_MIN;
    int tmp_end = mid + 1;
    for (int i = mid + 1; i <= right; i++) {
        sum += arr[i];
        if (sum > right_sum) {
            right_sum = sum;
            tmp_end = i;
        }
    }

    start = tmp_start;
    end = tmp_end;
    return left_sum + right_sum;
}

```

- 분할 정복 알고리즘은 배열을 좌우로 나누어 각각의 최대 구간 합을 계산한 뒤, 중앙을 포함하는 최대 구간 합과 비교하여 세 개의 값 중 제일 큰 값을 구하는 방식으로 동작한다.
- max_subarraySum 함수를 통해 구한 sum값이 max_sum값보다 크다면 최대 구간 합을 갱신하고 해당 영역의 직사각형의 좌표값을 저장한다.
- 좌우로 나누어 각각의 최대 구간 합을 구할 때, 해당 구간을 다시 반으로 나누어 계산하기 때문에 분할 정복이 이루어진다. 이를 통해 2D 행렬에서 최대 구간 합을 가지는 직사각형의 좌표 값을 효율적으로 구할 수 있다.

1.3. Algorithm5: 시간 복잡도 $O(n^3)$

```

// Algorithm 5:  $O(n^3)$  using Kadane's Algorithm
int MaxSumSubrectangle(vector<vector<int>>& mat, int& k, int& i, int& l, int& j) {
    int n = mat.size();
    int max_sum = INT_MIN;

    for (int left = 0; left < n; left++) {
        vector<int> tmp(n, 0);
        for (int right = left; right < n; right++) {
            for (int row = 0; row < n; row++) {
                tmp[row] += mat[row][right];
            }

            int start, end;
            int sum = kadane(tmp, start, end, n);

            if (sum > max_sum) {
                max_sum = sum;
                k = start;
                i = left;
                l = end;
                j = right;
            }
            else if (sum == max_sum) {
                if (left < i || (left == i && right < j) || (left == i && right == j && start < k) || (left == i && right == j && start == k && end < l)) {
                    k = start;
                    i = left;
                    l = end;
                    j = right;
                }
            }
        }
    }

    return max_sum;
}

```

- 해당 알고리즘은 Algorithm4와 마찬가지로 2D 배열을 먼저 1D 배열로 변환하는 것은 동일하지만, 1D 배열에서 분할 정복 기법이 아닌 kadane's Algorithm 기법을 사용해 최대 구간 합을 계산한다.

```
int kadane(vector<int>& arr, int& start, int& end, int n) {
    int max_sum = INT_MIN, current_sum = 0;
    start = -1;
    int tmp_start = 0;
    for (int i = 0; i < n; i++) {
        current_sum += arr[i];
        if (current_sum > max_sum) {
            max_sum = current_sum;
            start = tmp_start;
            end = i;
        }

        if (current_sum < 0) {
            current_sum = 0;
            tmp_start = i + 1;
        }
    }

    return max_sum;
}
```

- kadane's Algorithm은 연속된 부분 구간 중에서 최대 합을 빠르게 찾는 알고리즘으로, 각 1D 배열에 대해 이를 적용하여 최대 구간 합을 찾고 이를 통해 2D 행렬 내에서 최대 직사각형의 합을 구한다.
- max_sum은 현재까지 발견된 최대 구간 합을 저장하는 변수다. 초기값으로 INT_MIN을 사용하여 어떤 배열이라도 첫 번째 부분 배열의 합이 최대 구간 합이 될 수 있도록 설정한다.
- current_sum은 현재까지의 연속 부분 배열의 합을 저장하는 변수로, 각 배열의 요소를 순차적으로 더해가며 계산한다.
- start와 end는 최댓값을 갖는 부분 배열의 시작 인덱스와 끝 인덱스를 저장한다.
- tmp_start는 새로운 부분 배열이 시작될 때 그 시작 인덱스를 저장하는 임시 변수다.
- current_sum이 max_sum보다 클 경우, 최대 합을 갱신하고 그 때의 시작 인덱스와 끝 인덱스를 기록한다. 만약 current_sum이 0보다 작은 경우 부분 배열의 합이 음수가 되어 더 이상 연속된 부분 배열이 최대 합을 만들 가능성이 없다. 따라서 새로운 시작 부분 배열을 시작해야 하며 이를 위해 current_sum을 0으로 초기화하고 tmp_start를 현재 인덱스의 다음 인덱스로 이동시킨다.

2. Actual Running Time

[실험 환경]

OS: Windows 11 Home

CPU: Intel(R) Core(TM) i5-1240P @ 1.70GHz

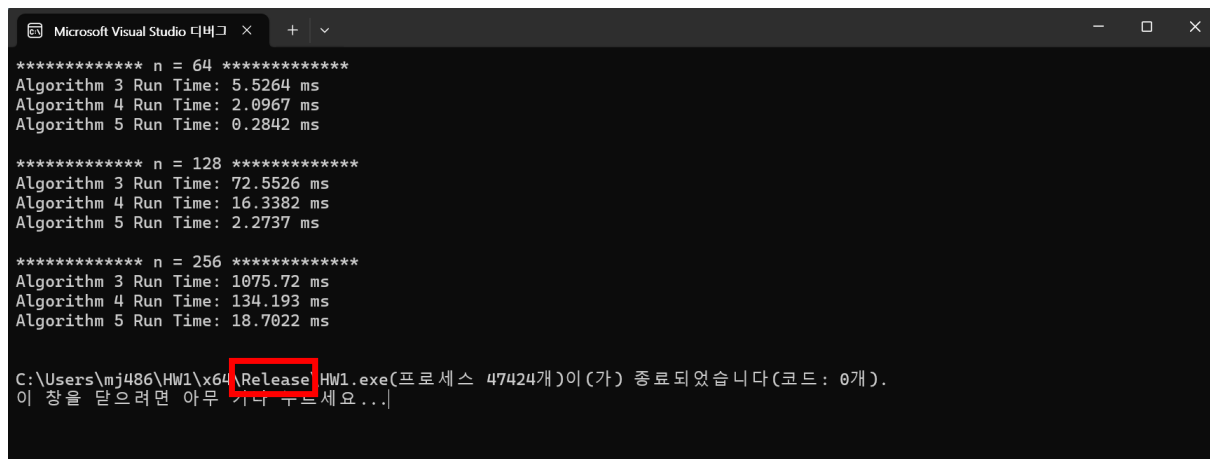
RAM: 16.00GB

Compiler: Visual Studio 22 Release Mode/x64 Platform

- cartoon, moon 두 부류의 data에 대해 실제 수행 시간을 측정하였고, 5회 측정 시간의 평균 값을 계산하였다.

- n=1024 or 2048인 경우 수행 시간이 10분을 초과하여 시간 관계 상 3회 측정의 평균 값을 계산하였고, n=2048일 때 Algorithm3번의 경우 수행 시간이 매우 길어 측정이 불가하였다.

1.1. cartoon



```
Microsoft Visual Studio 디버그 콘솔
***** n = 64 *****
Algorithm 3 Run Time: 5.5264 ms
Algorithm 4 Run Time: 2.0967 ms
Algorithm 5 Run Time: 0.2842 ms

***** n = 128 *****
Algorithm 3 Run Time: 72.5526 ms
Algorithm 4 Run Time: 16.3382 ms
Algorithm 5 Run Time: 2.2737 ms

***** n = 256 *****
Algorithm 3 Run Time: 1075.72 ms
Algorithm 4 Run Time: 134.193 ms
Algorithm 5 Run Time: 18.7022 ms

C:\Users\mj486\HW1\x64\Release\HW1.exe(프로세스 47424개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

[사진1. cartoon 시간 측정 1회차]

1) n=64

(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회	5.5264	2.0967	0.2842
2회	4.8833	2.0012	0.2701
3회	7.2808	3.1806	0.5929
4회	6.3615	2.3325	0.3021
5회	5.8106	2.4829	0.3013
평균	5.97252	2.41878	0.35012

2) n=128

(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회	72.5526	16.3382	2.2737
2회	71.0724	23.835	2.1986
3회	85.2258	21.0684	2.8888
4회	79.0147	16.28	2.3624
5회	71.4882	16.6404	2.2009
평균	75.87074	18.8324	2.38488

3) n=256

(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회	1075.72	134.193	18.7022
2회	1175.39	147.237	18.4152
3회	1160.89	137.247	19.0206
4회	1151.69	136.008	23.2231
5회	1109.3	137.368	18.7358
평균	1134.598	138.4106	19.6194

4) n=512

(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회	17312.5	1097.34	178.861
2회	17358.3	1104.15	176.234
3회	18653.5	1099.02	163.379
4회	16101.7	1056.05	155.853
5회	16230.7	1058.12	156.498
평균	17131.34	1082.936	166.165

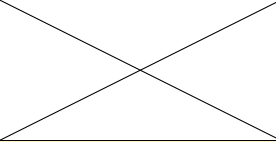
5) n=1024

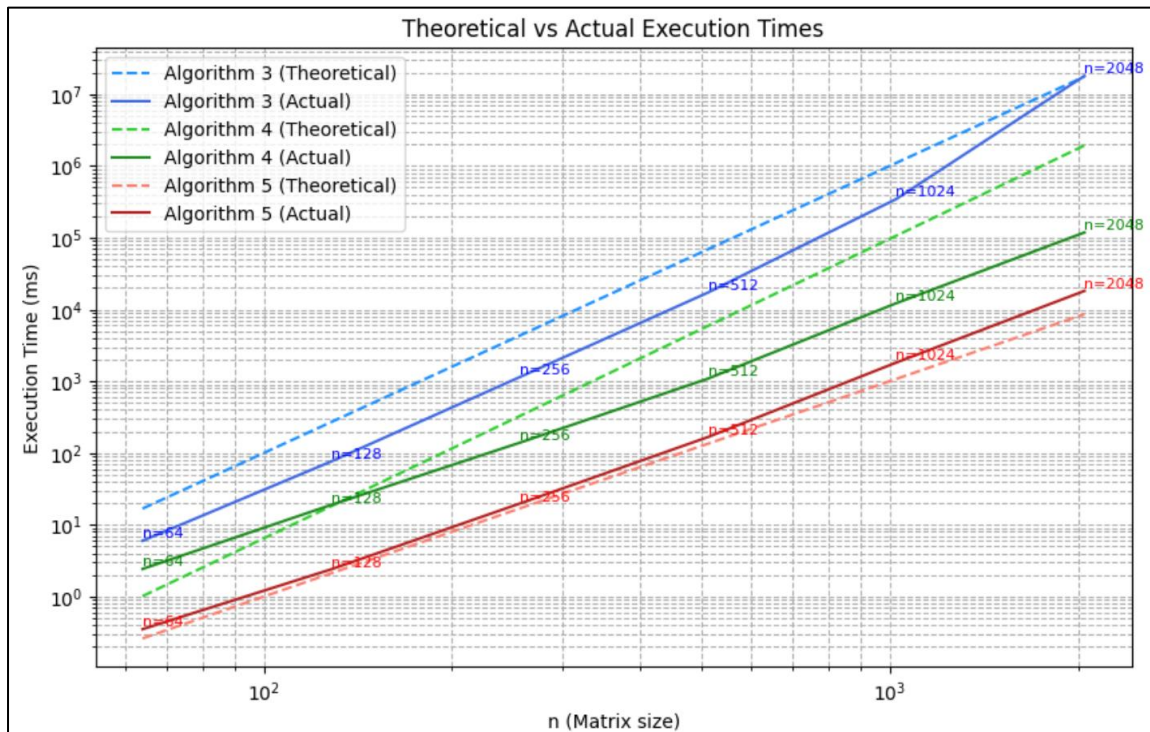
(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회	400336	10918.3	1843.44
2회	368268	16552.7	2175.67
3회	266976	9138.86	1427.09
평균	345193.3	12203.29	1815.4

6) n=2048

(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회		129775	16560.1
2회		148776	23050.6
3회		79743.8	14815.1
평균	측정 불가	118431.6	18141.93



[사진2. 이론적인 수행 시간 vs 실제 수행 시간 - cartoon]

- 파이썬의 matplotlib.pyplot을 이용해 이론적인 수행 시간과 실제 수행 시간 간의 관계를 그래프로 그려보면 위 사진과 같다.
- 이론적인 수행 시간은 CPU가 초당 1억번의 연산을 수행한다고 가정하고, 연산 횟수를 1억으로 나누어 계산하였다.

[변환 식]

$$\text{초 단위 시간} = \frac{\text{연산 횟수}}{10^9}$$

[예시]

- 예를 들어, Algorithm3의 시간 복잡도가 n⁴일 때 n=64인 경우 n⁴=16,777,216가 되므로 이를 1억으로 나눈 후, 1000을 곱해 ms 단위로 나타내면 약 16.78ms가 된다.

$$\frac{16,777,216}{10^9} \approx 0.0168s \approx 16.78ms$$

- n=2048일 때 Algorithm3의 실제 수행 시간은 측정 불가했던 관계로 이론적인 수행 시간과 동일하게 설정하였다.
- 그래프에 대한 분석은 아래에서 진행하겠다.

1.2. moon

1) n=64

(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회	4.9163	1.7585	0.3147
2회	6.2069	2.6772	0.2892
3회	4.8973	2.0648	0.3164
4회	5.0884	1.9193	0.3153
5회	5.8377	1.8658	0.2923
평균	5.38932	2.05712	0.30558

2) n=128

(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회	76.7416	15.4628	2.2405
2회	69.3174	15.2282	2.2181
3회	74.6408	15.2973	2.3793
4회	72.5767	14.9146	2.2433
5회	72.7215	16.2493	2.2347
평균	73.1996	15.43044	2.26318

3) n=256

(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회	1138.39	136.039	18.3025
2회	1109.72	127.15	22.2039
3회	1101.38	130.887	19.3553
4회	1099.7	129.176	18.9871
5회	1117.96	128.934	18.6266
평균	1113.43	130.4372	19.49508

4) n=512

(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회	16992.3	1090.5	162.464
2회	17722	1083.93	162.997
3회	18412.5	1081.83	163.392
4회	16088.2	1045.55	156.013
5회	17135	1153.55	168.781
평균	17270	1091.072	162.729

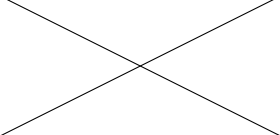
5) n=1024

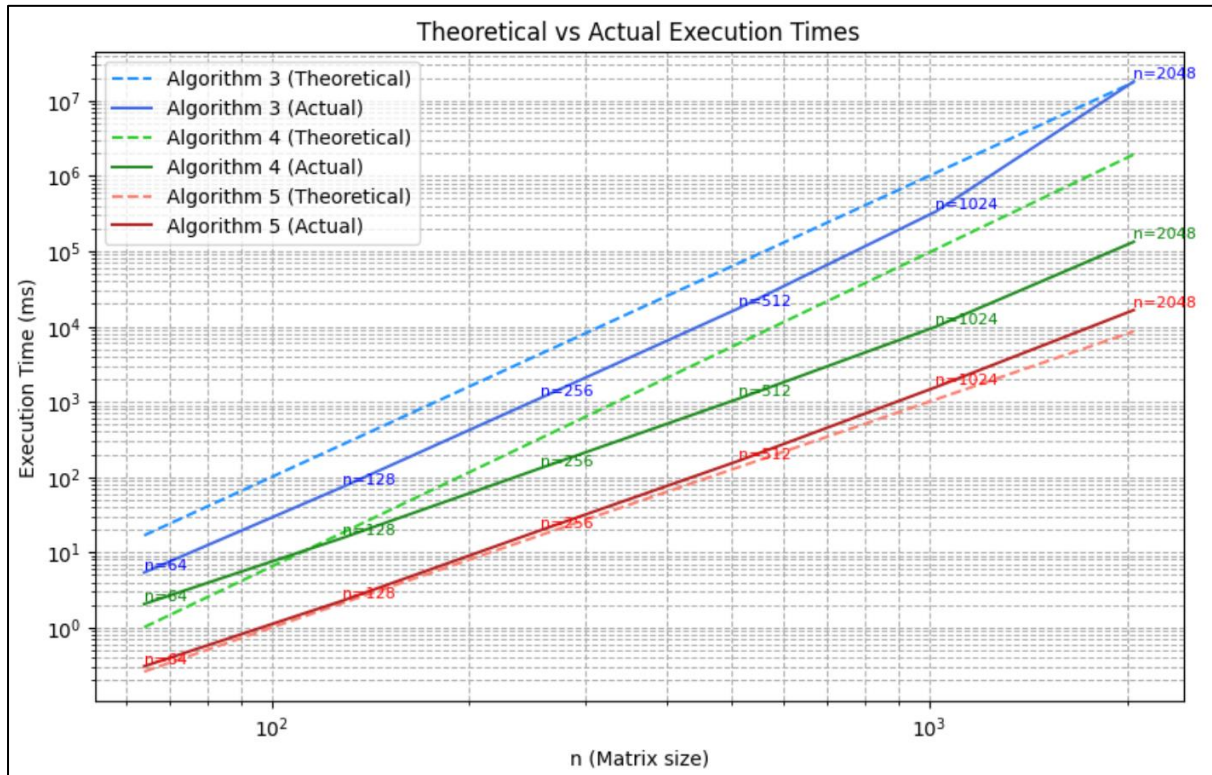
(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회	284518	9366.87	1559.43
2회	371008	9454.59	1547.87
3회	355099	11003.4	1582.22
평균	336875	9941.62	1563.173

6) n=2048

(단위: ms)

	Algorithm3	Algorithm4	Algorithm5
1회		150988	17180.1
2회		122447	16147.5
3회		126936	16275.2
평균	측정 불가	133457	16534.3



[사진3. 이론적인 수행 시간 vs 실제 수행 시간 – moon]

- moon도 실제 수행 시간이 cartoon과 비슷하게 측정되기 때문에 매우 유사한 그래프 형태를 가진다.
- Algorithm3의 경우, 이론적으로 가장 높은 시간 복잡도를 가지며 실제로도 가장 느린 성능을 보인다. 특히 n 값이 커질수록 실제 수행 시간이 기하급수적으로 증가하는 것을 그래프를 통해 확인할 수 있었다. (해당 그래프에서는 y축 증가 단위가 10의 지수 승 단위이므로 그 변화가 미미해 보이지만 실제로 n 값의 증가에 따른 수행 시간 증가량은 매우 크다.)
- Algorithm4는 이론적으로 $O(n^3 \log n)$ 의 시간 복잡도를 가지지만 n 값이 작은 경우에는 Algorithm5와 성능 차이가 크지 않다. 그러나 n 이 커질수록 $\log(n)$ 의 영향을 받아 성능 차이가 발생하게 된다. 그래프를 보면 실제로도 n 이 커질수록 성능이 감소하지만 Algorithm3보다는 훨씬 효율적으로 동작하는 것을 알 수 있다.
- 마지막으로 Algorithm5는 이론적으로 가장 낮은 시간 복잡도를 가지고 있으며, 실제로도 가장 빠른 성능을 보인다. 특히 n 이 커질수록 다른 알고리즘과의 성능 차이가 더 명확해진다. 이는 kadane's Algorithm이 매우 효율적이라는 사실을 입증한다.
- 또한, 해당 그래프를 확인해보면 Algorithm5는 n 이 커질수록 이론적인 수행 시간과 실제 수행 시간이 엇비슷하지만 Algorithm3, 4의 경우 n 값이 커짐에 따라 Algorithm5에 비해 더 큰 차이가 발생하는 것을 알 수 있다.
- 그렇다면 이러한 이론적인 시간 복잡도와 실제 성능에 차이가 발생하는 이유는 무엇인가? 실제

성능을 분석할 때, 상수 항과 연산 환경의 영향을 무시할 수 없기 때문이다. 예를 들어 Algorithm4와 5는 이론적으로 차이가 있지만, n 의 값이 작을 때에는 실제 성능 차이가 거의 없다. 이는 상수 항이나 메모리 접근 방식과 같은 세부적인 최적화 요소들도 성능 분석에 중요한 역할을 할 수 있다는 것을 보여준다.

- 충분히 큰 n 에 대하여 $O(n^4)$ 방법과 $O(n^3 \log n)$ 방법의 차이를 확인하는 가장 명확한 방법은 충분한 시간을 들여 실제 성능 테스트를 통해 실행 시간을 비교하고 이를 그래프로 시각화하는 것이다. 이론적으로 $O(n^4)$ 는 n 이 2배로 커질 때마다 실행 시간이 약 16배 증가할 것이고, $O(n^3 \log n)$ 은 n 이 2배로 커질 때마다 실행 시간이 약 8배보다 조금 작은 정도로 증가할 것이다.

- 그래프로 실행 결과를 확인하면 두 알고리즘 간의 차이가 점점 더 크게 나타나는 지점을 파악할 수 있다. 이는 알고리즘의 이론적 복잡도 차이를 확인할 수 있는 좋은 방법이다. 이를 통해 이론적으로 $O(n^4)$ 가 매우 비효율적임을 확인하고, $O(n^3 \log n)$ 알고리즘이 더 큰 n 에서도 보다 효율적이고 훨씬 빠른 속도로 수행되는 것을 시각적으로 확인할 수 있다.

*Python code for graph

```
import matplotlib.pyplot as plt
# cartoon
# n 값
n_values = [64, 128, 256, 512, 1024, 2048]

# 이론적인 수행 시간
theoretical_times = {
    'Algorithm 3': [16.78, 268.44, 4295.00, 68719.48, 1099511.63, 17592186.04],
    'Algorithm 4': [1.01, 18.06, 324.83, 5865.22, 106300.44, 1929704.47],
    'Algorithm 5': [0.26, 2.10, 16.78, 134.22, 1073.74, 8589.93]
}

# 실제 수행 시간
actual_times = {
    'Algorithm 3': [5.97252, 75.87074, 1134.598, 17131.34, 345193.3, 18000000],
    'Algorithm 4': [2.41878, 18.8324, 138.4106, 1082.936, 12203.29, 118431.6],
    'Algorithm 5': [0.35012, 2.38488, 19.6194, 166.165, 1815.4, 18141.93]
}

# 그래프 그리기
plt.figure(figsize=(10, 6))

# Algorithm 3
plt.plot(n_values, theoretical_times['Algorithm 3'], label='Algorithm 3 (Theoretical)', linestyle='--', color='dodgerblue')
plt.plot(n_values, actual_times['Algorithm 3'], label='Algorithm 3 (Actual)', linestyle='-', color='royalblue')

# Algorithm 4
plt.plot(n_values, theoretical_times['Algorithm 4'], label='Algorithm 4 (Theoretical)', linestyle='--', color='limegreen')
plt.plot(n_values, actual_times['Algorithm 4'], label='Algorithm 4 (Actual)', linestyle='-', color='forestgreen')

# Algorithm 5
plt.plot(n_values, theoretical_times['Algorithm 5'], label='Algorithm 5 (Theoretical)', linestyle='--', color='salmon')
plt.plot(n_values, actual_times['Algorithm 5'], label='Algorithm 5 (Actual)', linestyle='-', color='firebrick')
```

```

# n값 표시
for i, n in enumerate(n_values):
    plt.text(n, actual_times['Algorithm 3'][i], f'n={n}', fontsize=8, color='blue', verticalalignment='bottom')
    plt.text(n, actual_times['Algorithm 4'][i], f'n={n}', fontsize=8, color='green', verticalalignment='bottom')
    plt.text(n, actual_times['Algorithm 5'][i], f'n={n}', fontsize=8, color='red', verticalalignment='bottom')

# 그래프 설정
plt.xscale('log')
plt.yscale('log')
plt.xlabel('n (Matrix size)')
plt.ylabel('Execution Time (ms)')
plt.title('Theoretical vs Actual Execution Times')
plt.legend()
plt.grid(True, which="both", ls="--")

# 그래프 출력
plt.show()

```

- moon의 경우 위 코드에서 실제 수행 시간의 값만 변경해주면 된다.