1. **Algorithm**
   1. **Algorithm 3**

본인의 ‘Algorithm 3’ 알고리즘은 완전 탐색, 즉 프루트포스 알고리즘으로

2차원 배열에서 가능한 모든 Subrectangle을 탐색하면서 각 직사각형내 광도 값의 합을 구하여

최대 부분합을 찾는 방식으로 구현됐고 과정을 다음과 같다.

먼저 본인은 배열 내의 누적합을 계산하여 Summed Area-Table을 만들어서

나중에 Subrectangle을 탐색할 때, 부분합을 편하게 계산할 수 있도록 했다.

스크린샷, 텍스트, 폰트이(가) 표시된 사진

자동 생성된 설명

그 다음 상단, 하단, 좌측, 우측 인덱스가 이동할 수 있는 모든 경우에 대해서 이동하면서

모든 Subrectangle을 탐색했다. 이때 본인이 구현한 Table의 각 인덱스에 저장된 값들은 모두 좌측 상단의

인덱스가 (0,0)인 기준으로 계산된 누적합이기 때문에 추가적인 계산과정이 필요했다. 따라서 아래의 그림과

같이 좌측 Subrectangle의 누적합을 제거하고, 상단 Subrectangle 누적합을 제거한 다음 두 번 제거된

좌 상단 대각선의 Subrectangle의 누적합을 더해줌으로써 원하는 크기의 누적합을 구할 수 있었다.

마지막으로 모든 Subrectangle을 탐색하면서 최대 누적합을 찾으면 그 합과

그 누적합을 이루는 사각형의 각 인덱스를 저장했다.

텍스트, 라인, 도표, 스크린샷이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

위의 두 코드를 보면 알 수 있듯이 각 반복문의 시간 복잡도는 T**(n) = n2, T(n) = n4** 이며,

따라서 최종적으로 이 알고리즘의 시간복잡도는 T**(n) = n2 + n4** 🡪 **O(n4)**이다.

* 1. **Algorithm 4**

본인의 ‘Algorithm 4’ 알고리즘은 열을 기준으로 누적합을 계산 후 그 열에 대해

Divide-and-Conquer 알고리즘을 적용하여 최대 부분합을 구하는 방식으로 구현됐고 과정은 다음과 같다.

먼저 배열의 각 열에 대해서 누적합을 계산하여 좌표에서 해당 열까지의 합을

빠르게 계산할 수 있도록 했다. 그 다음 분할정복 알고리즘을 통해 최대 누적합을 계산하는 함수를

호출하여, 최대 누적합과 그 누적합을 이루는 사각형의 각 인덱스를 저장했다.

모든 열의 누적합을 계산할 수 있도록 i와 j인덱스를 이용해 2차 for문으로 인덱스를 탐색했다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 라인, 평행이(가) 표시된 사진

자동 생성된 설명

열 별로 구한 1차원 배열 tmp에 대해서는 분할정복 알고리즘을 적용하여 최대 누적합을 찾았다.

최상단에는 배열의 원소가 하나가 남았을 때, 그 값을

반환하는 base case를 조건문으로 넣어 주었고,

그 밑에는 배열을 반으로 나누어 각각 재귀적으로

최대 부분합을 찾을 수 있도록 했다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

중앙을 기준으로 좌우로 확장해 가면서 중앙을 가로지르는

부분에서의 최대 부분합도 계산했다.

아래의 그림을 보면 그 과정에 대한 이해가 더 쉬울 것이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

모든 과정을 마친 뒤 좌측 구간, 우측 구간 그리고

중앙 구간의 최대부분합을 비교하여 가장 큰 값을

반환했고, 이때 최대 부분합의 양 끝 인덱스를 a와 b에

저장했다.

배열을 반으로 나누어 각각에 대해 MaxSubSum 함수를 호출하는 과정과

중앙 구간의 부분합을 구하는 과정의 시간 복잡도를 합하면 T(n) = 2T(n/2) + O(n) 이다.

그리고 재귀 관계식의 일반 형태는 다음과 같다. T(n) = a(T/b) + O(nd)

*\* 중앙 구간의 최대 부분합을 구하기 위해서는 최악의 경우 모든 원소를 탐색해야 할 수 있기 때문에*

*Big-O 시간 복잡도를 적용하였다.*

*\* a는 부분 문제의 개수, b는 문제를 나누는 비율, 그리고 O(nd)는 나눠진 문제를 합치는 데 걸리는 시간*

이때, a = bd인 경우 시간 복잡도는 O(nd logn)이며, 본인의 알고리즘은 a = 2, b = 2, d = 1이므로

적용된다. 따라서 MaxSubSum 함수의 시간 복잡도는 O(n log n)이다.

열 별로 1차원 배열을 만들고, ( T(n) = n2)

만들어진 1차원 배열을 누적합 배열로 계산한 뒤 (T(n) = n)

누적합 배열에서 최대 부분합을 구하기 위해 MaxSubSum 함수를 호출하기 때문에 ( T(n) = nlogn)

최종적으로 이 알고리즘의 시간 복잡도는 **T(n) = n2(n+ nlogn) 🡪 O(n3logn)**이다.

* 1. **Algorithm 5**

이 알고리즘은 algorithm 4와 마찬가지로 각 열 별로 누적합을 구하는 것은 같지만

계산된 열에 대해서 Kadane 알고리즘을 적용한 부분이 다르다.

누적합을 구하는 구하는 부분은 위에서 설명했기 때문에 Kadane 알고리즘에 대해서만 설명해보겠다.

텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

자동 생성된 설명

Kadane 알고리즘은 동적 프로그래밍 기법을 적용하여 열의 처음부터 끝까지 추적하여 최대 누적합을

구한다. 현재까지 계산된 부분합과 이전까지의 최대 부분합을 비교하며,

현재 요소를 포함할지 아니면 새로 시작할지를 결정하면서 최종적으로 최대 부분합을 반환한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명 텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

먼저 sum에 현재 tmp 배열 값을 더해 sum을 갱신한다. 만약 sum이 음수라면 부분합을 0으로 초기화하고,

시작점도 갱신한다. 또는 sum이 현재 최대 누적합 maxSum보다 클 경우 maxSum을 sum으로 갱신하고,

그때의 양끝 인덱스를 저장한다. 만약 모든 값이 음수가 아니라면 그대로 maxSum을 반환하고,

모든 값이 음수라면 가장 큰 음수를 찾아서 반환한다.

코드를 보면 간단하게 알 수 있듯이 kadane 알고리즘의 시간복잡도는 T(n) = n + n 🡪 O(n) 이다.

따라서 처음에 열을 기준으로 1차원 배열을 만들고 (T(n) = n2),

만들어진 배열을 통해 누적합 배열을 만든 뒤(T(n) = n) 각 열에 대해 kadane 함수를 호출하여

T(n) = n이 걸리기 때문에 이 알고리즘의 시간복잡도는 **T(n) = n2(n+ n)🡪O(n3)**이다.

1. **Time**

**-실험 목적**: 각 알고리즘이 다양한 크기의 입력에 대해 어떻게 성능을 발휘하는지 측정한 뒤

이론적인 시간 복잡도와 수행시간과의 관계에 대해서 분석한다.

**-실험 환경:** ARM64 플랫폼에서 Release 모드로 컴파일한 후, 동일한 입력 데이터에 대해

여러 차례 반복 실행하여 평균 시간을 측정해서 사용했다.

**-결과 분석:**

\*아래는 moon.pgm 파일에 대한 각 케이스 별로 실행시간을 구한 표이다.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Algorithm | n | Case 1(ms) | Case 2(ms) | Case 3(ms) | Case 4(ms) | Case 5(ms) | Average |
| Algorithm 3 | 64 | 7.4022 | 8.0034 | 7.4031 | 9.0025 | 6.7028 | 7.7028 |
| 128 | 150.413 | 196.837 | 92.691 | 125.962 | 82.797 | 129.74 |
| 256 | 1645.842 | 1141.136 | 1133.916 | 1216.381 | 1131.225 | 1253.7 |
| 512 | 17777.149 | 17852.201 | 17472.182 | 17546.019 | 17548.232 | 17.689(s) |
| 1024 | 278124.112 | 285086.132 | 294438.321 | 289436.179 | 293265.256 | 288.07(s) |
| 2048 | - | - | - | - | - | - |
| Algorithm 4 | 64 | 6.0075 | 4.2043 | 5.0131 | 5.2023 | 5.1038 | 5.1062 |
| 128 | 64.825 | 25.541 | 15.597 | 17.004 | 17.703 | 27.934 |
| 256 | 254.427 | 187.169 | 142.797 | 140.429 | 142.928 | 173.55 |
| 512 | 1766.496 | 1195.331 | 1158.192 | 1192.743 | 1232.238 | 1.3090(s) |
| 1024 | 9898.147 | 9970.552 | 9679.982 | 9763.210 | 9609.609 | 9.7873(s) |
| 2048 | 98418.185 | 98035.002 | 96599.082 | 95740.184 | 95970.209 | 96.952(s) |
| Algorithm 5 | 64 | 1.0323 | 0.9487 | 0.9947 | 0.9561 | 0.9452 | 0.9754 |
| 128 | 4.415 | 5.408 | 4.133 | 3.966 | 4.004 | 4.3852 |
| 256 | 70.660 | 68.094 | 130.181 | 39.675 | 56.605 | 73.043 |
| 512 | 406.398 | 388.129 | 342.511 | 289.841 | 279.921 | 0.34136(s) |
| 1024 | 3001.973 | 2483.209 | 2186.613 | 2301.342 | 2183.863 | 2.4314(s) |
| 2048 | 38927.021 | 33123.184 | 32134.211 | 32076.216 | 32042.178 | 33.660(s) |

*\*cartoon.pgm의 경우에는 아래의 표에만 평균 실행시간을 작성하였다.*

텍스트, 소프트웨어, 컴퓨터, 스크린샷이(가) 표시된 사진

자동 생성된 설명

\*위의 사진은 n=128에 대해 5번 실행한 결과이다.

텍스트, 소프트웨어, 컴퓨터, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

\*위의 사진은 n=64부터 n=2048까지 실행한 결과이다.

아래의 표는 위에서 진행한 실험 결과를 토대로 각 알고리즘 별 입력크기에 따른 평균 실행시간을

정리한 표이다.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 알고리즘  입력 크기 | **시간 단위** | **Algorithm 3** | | **Algorithm 4** | | **Algorithm 5** | |
| **pgm 파일** | / | moon | cartoon | moon | cartoon | moon | cartoon |
| **n=64** | ms | 7.7028 | 8.6486 | 5.1062 | 4.3934 | 0.9754 | 1.2286 |
| **n=128** | ms | 129.74 | 175.45 | 27.934 | 36.174 | 4.3852 | 5.628 |
| **n=256** | ms | 1253.7 | 1212.3 | 173.55 | 277.45 | 73.043 | 44.547 |
| **n=512** | s | 17.689 | 17.678 | 1.3090 | 1.3920 | 0.34136 | 0.3962 |
| **n=1024** | s | 288.07 | 301.73 | 9.7873 | 9.8433 | 2.4314 | 2.0820 |
| **n=2048** | s | - | - | 96.952 | 95.316 | 33.660 | 30.870 |

-이론적인 시간 복잡도와 실제 실행 시간과의 관계

**1. Algorithm 3 (O(n4)) 분석**

**- 수학적 분석** : 시간 복잡도는 다음과 같이 계산된다.

**n2** : 누적합 Table을 만드는 두개의 반복문

**n4** : 모든 서브 직사각형 내의 합을 계산하기 위한 또 다른 네 개의 반복문

따라서 총 시간 복잡도는 : **O(n4) = n2 + n4**

**- 실험 결과** : n = 64일 때 평균 시간은 약 7.7ms, n = 512일 때 약 17.7초, n = 1024일 때 약 288초로,

입력 크기 n이 증가할수록 **기하급수적으로 시간이 증가**하는 것을 확인할 수 있다. 이론적으로

예상한 O(n^4)의 복잡도가 실험에서도 잘 드러난다.

**- 연관성 분석** : Algorithm 3은 이론적으로 O(n4)의 성능을 보이며, 실험에서도 큰 입력에서는 **시간 초과**가

발생할 정도로 비효율적임을 확인할 수 있다. n이 두 배 증가할 때마다 수행 시간이

대략 **16배 증가**하는 경향이 있으며, 이는 O(n4) 복잡도를 충실히 반영한 결과이다.

**2. Algorithm 4 (O(n3 log n)) 분석**

**- 수학적 분석** : 분할 정복 알고리즘이 적용되며 시간 복잡도는 다음과 같이 계산된다.

**n2** : 열을 기준으로 모든 경우의 1차원 배열을 계산

**n**: 만들어진 1차원 배열을 통해 누적합 배열을 계산

**nlogn** : 모든 누적합 배열에서 분할 정복 알고리즘으로 최대 부분합을 계산

따라서 총 시간 복잡도는 : **O(n3 logn) = n2(n+ nlogn)**

**- 실험 결과** : n = 64일 때 평균 시간은 약 5.1ms, n = 512일 때 약 1.3초, n = 1024일 때 약 9.7초로,

O(n^3 log n) 복잡도가 반영된 결과를 볼 수 있다.

**- 연관성 분석**: 이론적으로 O(n^3 log n)의 성능을 보이며, 실제 실험 결과에서도 성능이 괜찮게 나타났다.

n이 증가할수록 시간이 조금 더 빠르게 증가하지만, **log n**이 포함되어 있기 때문에 O(n^4)보다

훨씬 효율적이다. 이론적으로 n이 두 배로 증가할 때 약 **10배 내외로 시간이 증가**하는 패턴을

보여야 하지만 실제 측정시간을 보면 그보다는 완만하게 시간이 증가하는 것을 볼 수 있다.

**3. Algorithm 5 (O(n^3)) 분석**

**- 수학적 분석**: Kadane 알고리즘을 적용되며 시간 복잡도는 다음과 같이 계산된다.

**n2** : 열을 기준으로 모든 경우의 1차원 배열을 계산

**n**: 만들어진 1차원 배열을 통해 누적합 배열을 계산

**n** : 모든 누적합 배열에서 kadane 알고리즘으로 최대 부분합을 계산

따라서 총 시간 복잡도는 : **O(n3) = n2(n+ n)**

**- 실험 결과** : n = 64일 때 평균 시간은 약 0.97ms, n = 512일 때 약 0.34초, n = 1024일 때 약 2.43초로,

**가장 효율적인 성능**을 보였다.

**- 연관성 분석**: 이론적으로 O(n^3)의 성능을 보이며, 실제 실험 결과에서도 매우 효율적인 성능을

확인할 수 있다. n이 두 배로 증가할 때 약 **8배 내외로 시간이 증가**하는 패턴을 보여야 하지만

실제 측정시간을 보면 그보다는 완만하게 시간이 증가하는 것을 볼 수 있다.

**\* 종합 분석**

실험을 통해 얻은 결과를 바탕으로, 각 알고리즘의 이론적인 시간 복잡도와 실제 수행 시간 간의 관계를

다음과 같이 요약할 수 있다.

**- Algorithm 3 (O(n^4))**:

n이 증가함에 따라 시간이 **기하급수적으로 증가**하며, 큰 입력(n ≥ 512)에서는 **사실상 비효율적**이다.

n = 2048과 같은 **큰 입력에서는 사용할 수 없는 수준**임을 확인했다.

**- Algorithm 4 (O(n^3 log n))**:

n이 증가할 때 **log n**의 오버헤드로 인해 시간이 빠르게 증가하지만, O(n4)보다는 훨씬 효율적이다.

특히 n ≤ 512에서는 O(n4)보다 훨씬 효율적이고, O(n3)과도 실행시간에 큰 차이를 보이지 않아

매우 합리적인 성능을 보여주었다.

- **Algorithm 5 (O(n^3))**:

세 가지 알고리즘 중 **가장 효율적**이며, 큰 입력(n ≥ 1024)에서도 **실행 가능한 수준**의 성능을 보인다.

특히 입력이 크면 클수록 다른 알고리즘들보다 훨씬 뛰어난 성능을 확인할 수 있었다.

- 이론적인 시간 복잡도와 실제 실행시간과의 **차이점**

이론적으로 본다면 Algorithm 3는 입력이 2배 커질 때마다 16배씩, Algorithm 4는 입력이 2배 커질 때마다

약 10배씩, Algorithm 5는 입력이 2배 커질 때마다 8배씩 실행시간이 증가해야 하지만

실제로 측정한 결과를 보면 이론보다 더 **완만하게 증가**하는 것을 볼 수 있었다.

그 이유로는 이론적으로 계산한 시간복잡도는 최악의 경우를 가정하고 계산을 한 것이고,

실제로 실행할 때마다 매번 최악의 경우를 겪지 않으므로 이런 차이점이 발생했다고 볼 수 있다.

**\* 결론**

이론적인 시간 복잡도와 실제 수행 시간 간의 연관성은 분명히 강하게 존재하지만 약간의 차이점도 보이며

실험 결과가 이를 뒷받침하고 있다. Algorithm 3가 가장 비효율적인 성능을 보였고, Algorithm 5가

가장 효율적인 성능을 보였으며 그 차이는 입력 크기가 커질수록 더욱 명확해졌다.

1. **Setting**

-실험환경

OS: macOS Sonoma 14.3.1

CPU: Apple M1

RAM: 8.00GB

-가상머신 실험환경

OS: Windows 11 Home

CPU: virt-7.2 1GHz

RAM: 4.00GB

Compiler: Visual Studio 22 Release Mode/ARM64 Platform