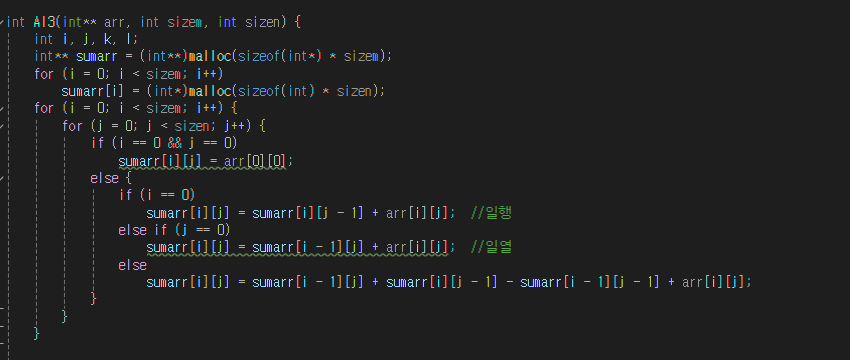
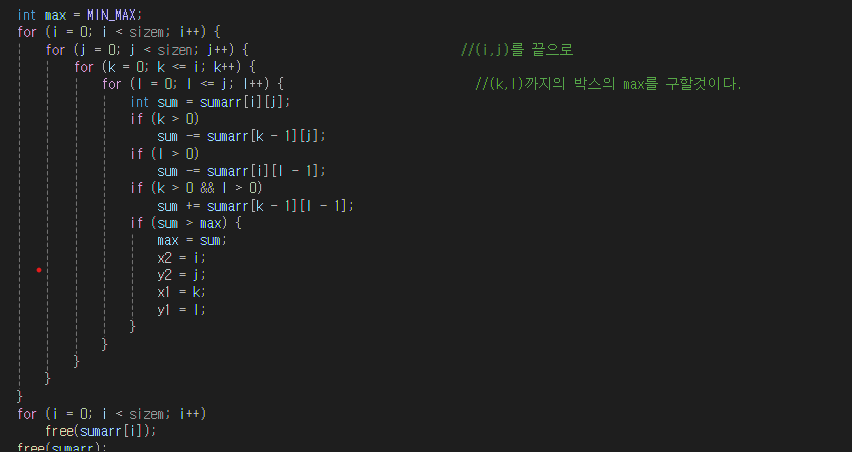
알고리즘설계와분석(임인성 교수님) HW1 보고서

20201223 김지오

1. 세 개 알고리즘 (Algorithm 3, Algorithm 4, 그리고 Algorithm 5) 각각에 대하여 자신이 사용한 방법을 몇 줄 이내의 문장으로 간략히 요약하라.

(1) Algorithm 3

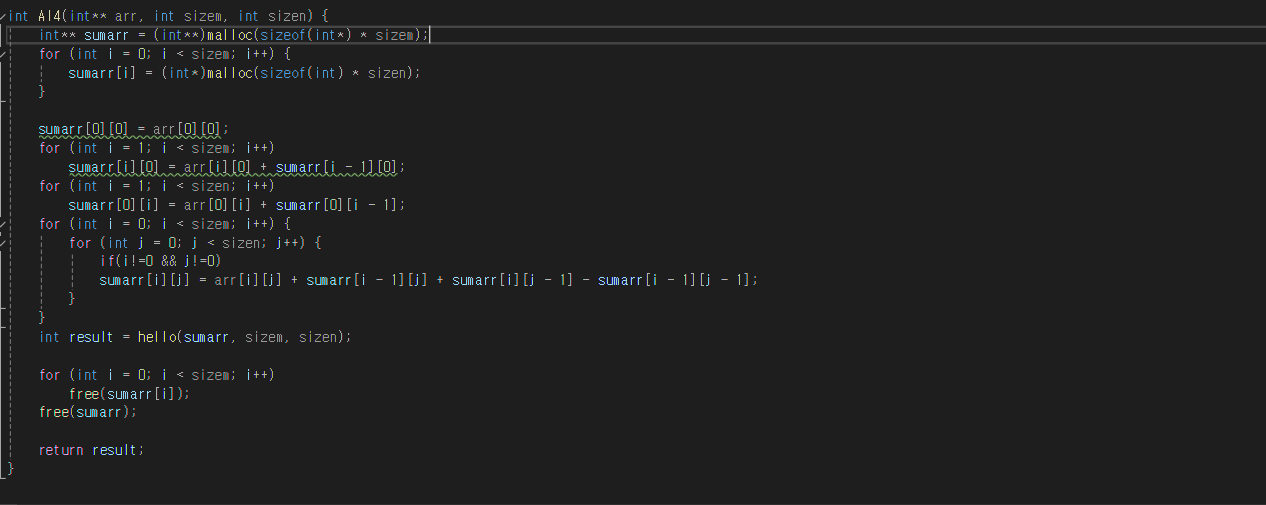


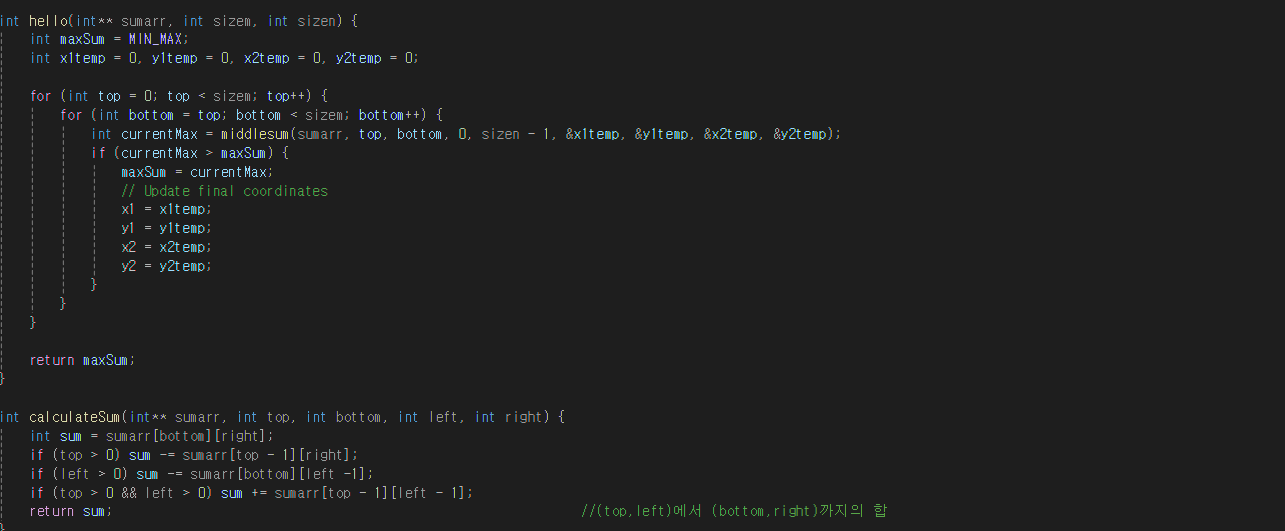


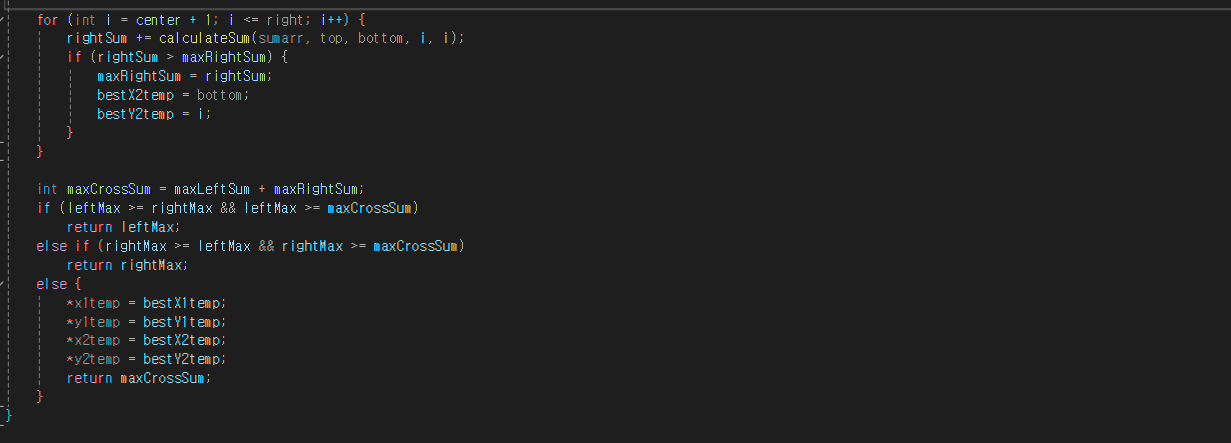
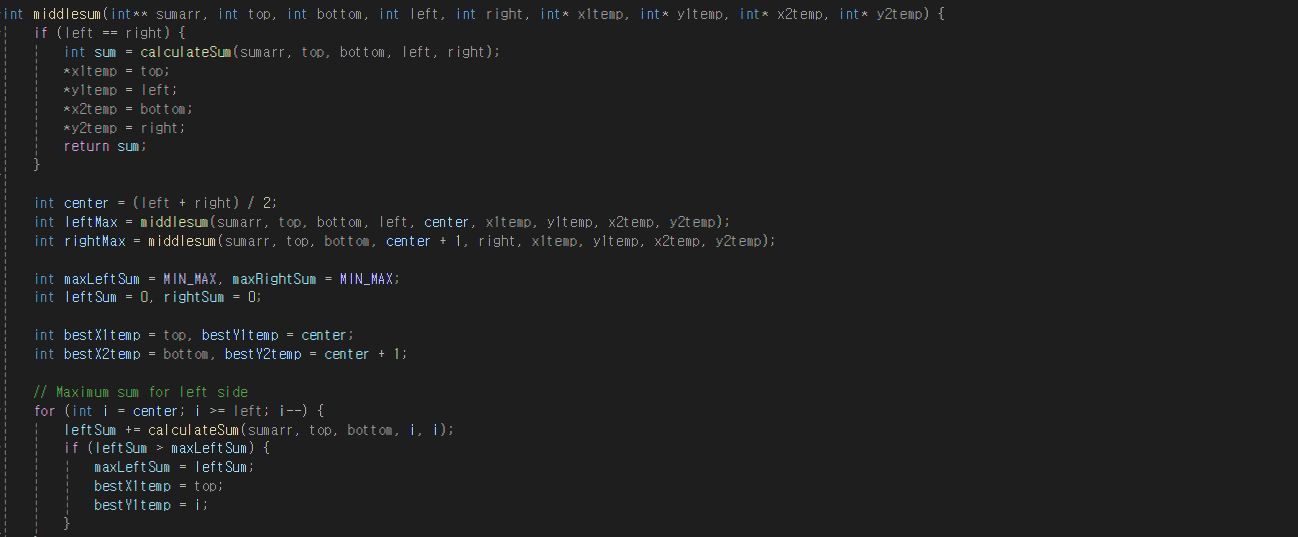
설명 : pgm파일을 이중배열 arr로 받았을 때, sumarr[i][j]는 sumarr[0][0]부터 sumarr[i][j]까지 네모 박스를 만들었을 때의 합이다. sumarr는 O(n^2)이 드는 것을 확인 할 수 있다.

그 뒤 (x1,y1)부터 (x2,y2)까지 드래그해서 나오는 박스의 넓이는 수업 시간에 교수님께서 설명하셨듯이 sumarr[x2,y2]에서 sumarr[x1][y2],sumarr[x2][y1]을 빼고, sumarr[x1][y1]을 더해주면 된다. 이 때 x2와 y2가 0이 될 때 예외처리를 해 준 모습이다. 그래서 모든 박스의 넓이를 구한 다음 max값을 찾는 방식이다. 그렇기 때문에 x1,y1,x2,y2를 0부터 n-1까지 찾기 때문에 O(n^4)이 된다. 이때 정확한 O(n^4)=O(sizem^2 \* sizen^2)으로 nxn으로 받기 때문에 O(n^4)으로 썼습니다.

(2) Algorithm 4







Al4 함수에서 Algorithm3와 똑같은 sumarr 2중배열을 만들어 줍니다. 이 때 Time complexity는 O(m\*n)=O(n^2)입니다. 그 다음, hello함수를 호출합니다.(알고리즘을 여러 번 바꾸다 보니 함수 이름이 마땅히 생각이 나지 않아 함수를 의미 없는 이름으로 설정하였습니다. 다음 번에 이런 사소한 것들도 고치겠습니다)

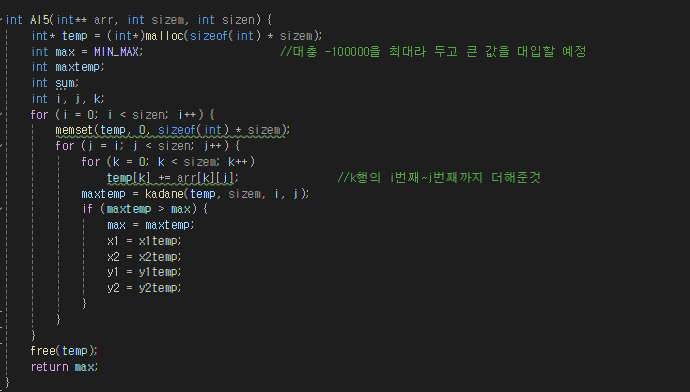
hello 함수는 top,bottom을 고정시키고 middlesum을 진행합니다. middlesum에서 받은 최댓값temp들을 비교하면서 최종적인 max값을 얻습니다. 이때 O(m^2\*O(middleSum))입니다.

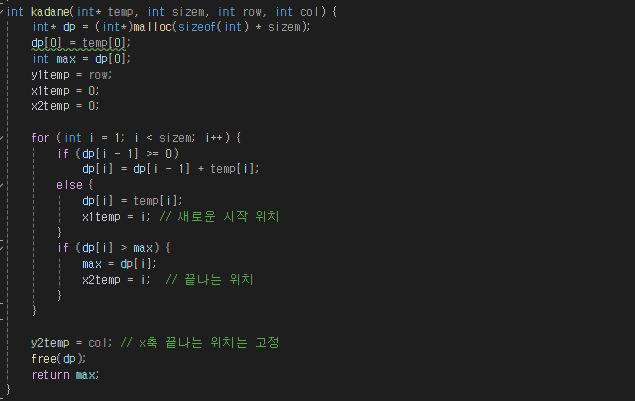
calculate sum 함수는 top,left,bottom,right을 받았을 때 (top,left)부터 (bottom,right)까지 드래그 하였을 때 생기는 사각형 안에 들어있는 arr값들의 합을 return하는 함수입니다. 이는 O(1)입니다.

middle sum 함수는 arr를 왼쪽,오른쪽 2갈래로 나누고 만약 박스가 중간을 지날 때를 처리해주는 역할을 합니다. 각각 leftMax,rightMax,maxMiddle중에 최대를 return합니다. 나누는 가짓수 logn, for문 n번해서 Time complexity는 O(n\*logn)이 됩니다.

최종적으로 O(m^2\*nlogn)이고, m=n이라 했을 때 O(n^3\*logn)이 됨을 알 수 있습니다.

(3) Algorithm 5





왼쪽 i열과 오른쪽 j열을 잡았을 때 temp[x]가 의미하는 것은 x행의 i열부터 j열까지 arr[][x]를 더하는 것입니다. 이 temp배열을 가지고 kadane 알고리즘을 이용해줍니다. 저희가 원하는 것은 temp배열에서 수업에서 배운 1차원 배열의 최대부분수열과 합을 구하는 것입니다. 그 다음, 최댓값일때의 x1temp,y1temp,x2temp,y2temp와 sum값을 잠시 저장해줍니다. 여기서 O(n)이 드는 것을 확인할 수 있습니다. 이 때 n은 sizem, 즉 열의 길이입니다. kadane 알고리즘을 벗어나서 i열부터 j열까지의 x1,y1,x2,y2의 temp와 max값을 받으면서 최대가 나오게 된다면 초기화 해줍니다. 이는 O(n^2)이며, 이때 n은 sizen즉 행의 길이입니다. 지금은 nxn으로 열의 길이와 행의 길이가 같기 때문에 O(n^3)이 되는 것을 확인할 수 있습니다.

2. 수행 시간 및 시간 복잡도의 관계를 충실히 분석할 수 있도록, 주어진 입력 크기 n에 대해 위 두 문제에 대한 실험을 진행하라. 시간 측정은 x64 플랫폼에서 Release 모드로 컴파일하여 ms 단위로 측정하라. 특히 실험 결과의 신뢰도를 높이기 위하여 가급적 동일한 입력 영상에 대하여 여러 번 수행한 후 평균 시간을 취할 것.

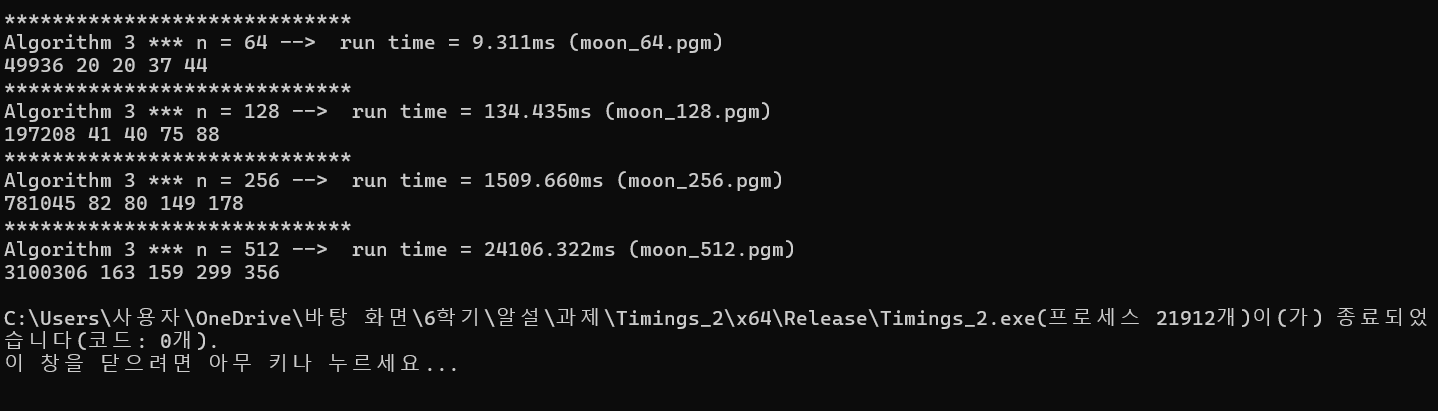
-3번의 결과에 같이 적겠습니다.

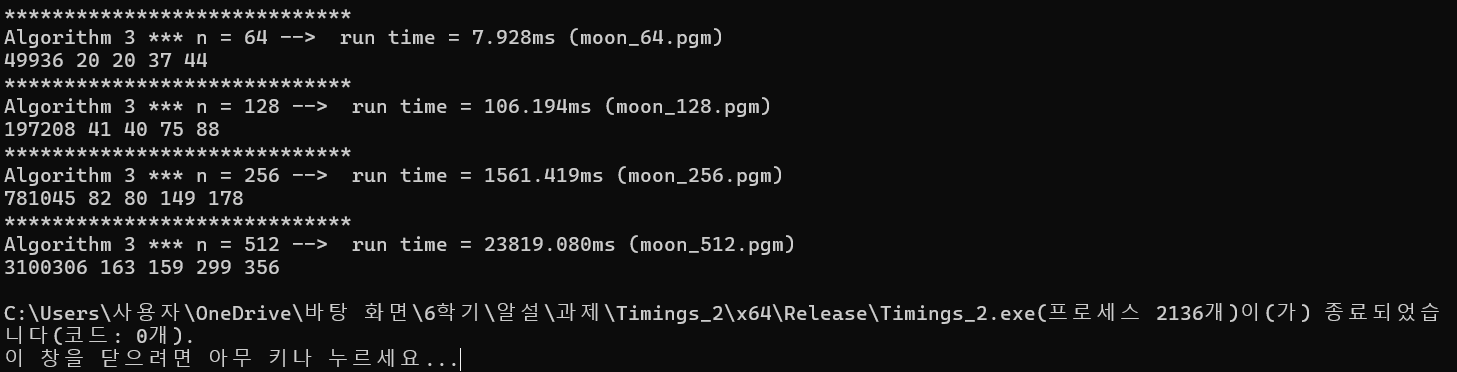
3. 상기 실험을 통하여 산출한 실험 결과로부터 세 알고리즘 (Algorithm 3, Algorithm 4, 그리고 Algorithm 5)의 이론적인 시간 복잡도와 수행 시간 간의 관계를 분석하고, 그로부터 자신이 발견한 사실과 그에 대한 의견을 보고서 파일에 명확히 기술하라. • 본 숙제의 중요한 목적 중의 하나는 이론적인 시간 복잡도와 실제 프로그램의 수행 시간과의 연관성을 찾아내는 것인데, 각 알고리즘에 대한 실험 결과를 적절한 형식의 테이블로 요약하라. 실제 측정한 시간 값과 이론적으로 구한 시간 복잡도의 연관성을 증명하기 위하여 그래프를 그린다던가 또는 수식으로 함수 관계를 보인다던가 하는 등의 공학적인 방법을 사용하여 자신의 주장을 논리적으로 기술할 것: 충분히 큰 n에 대하여 O(n 4 ) 방법과 O(n 3 log n) 방법의 차이를 어떻게 확인할 수 있을까? • 보고서에 실험 결과를 기술하기 전에 본인이 사용한 컴퓨터의 실험 환경을 정확히 기술할 것 (아래의 예를 참조할 것).

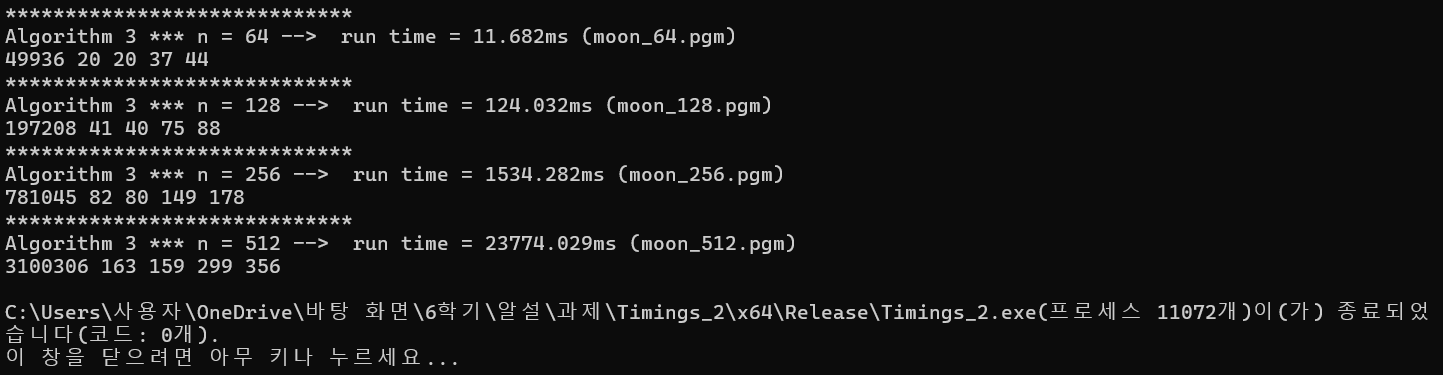
**moon.pgm의 수행 시간**

1024 X 1024부터 수행 시간이 길어지는 것을 확인 할 수 있습니다. Algorithm 밑의 숫자 4개는 각각 box의 최대 sum 값이며, 박스의 왼쪽 위 좌표를 (x1,y1), 오른쪽 밑 좌표를 (x2,y2)라 하였을 때 sum, x1, y1, x2, y2의 값입니다. (Algorithm3의 경우, 1024 x 1024, 2048 x 2048은 제외하였습니다.)

moon-Algorithm3







moon\_64.pgm의 평균 : (9.311+7.928+11.682)/3=9.64(ms)

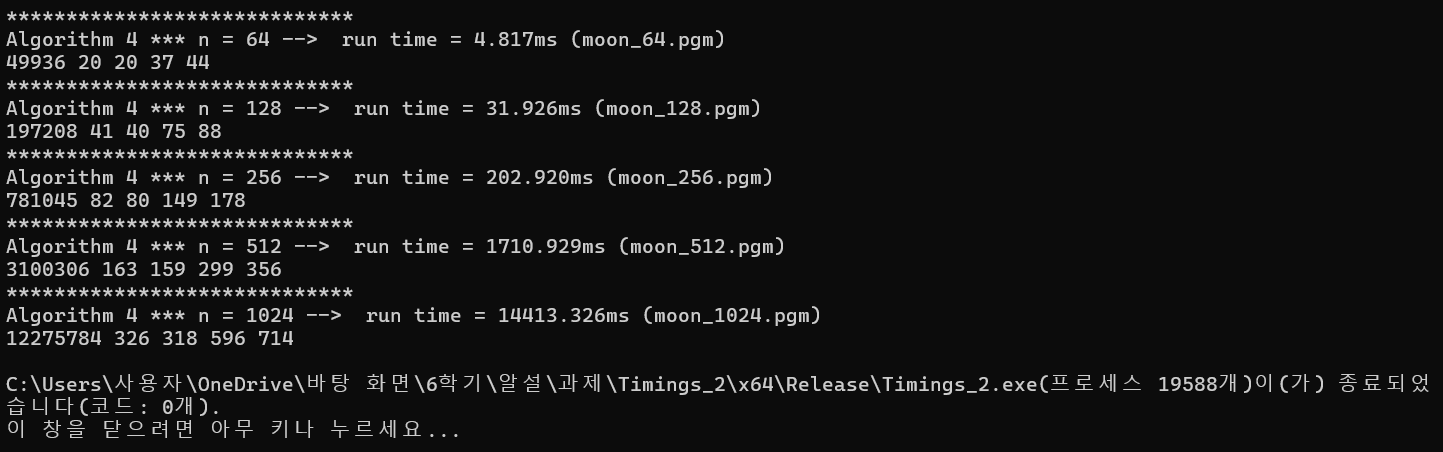
moon\_128.pgm의 평균 : (134.435+186.194+124.032)/3=147.22(ms)

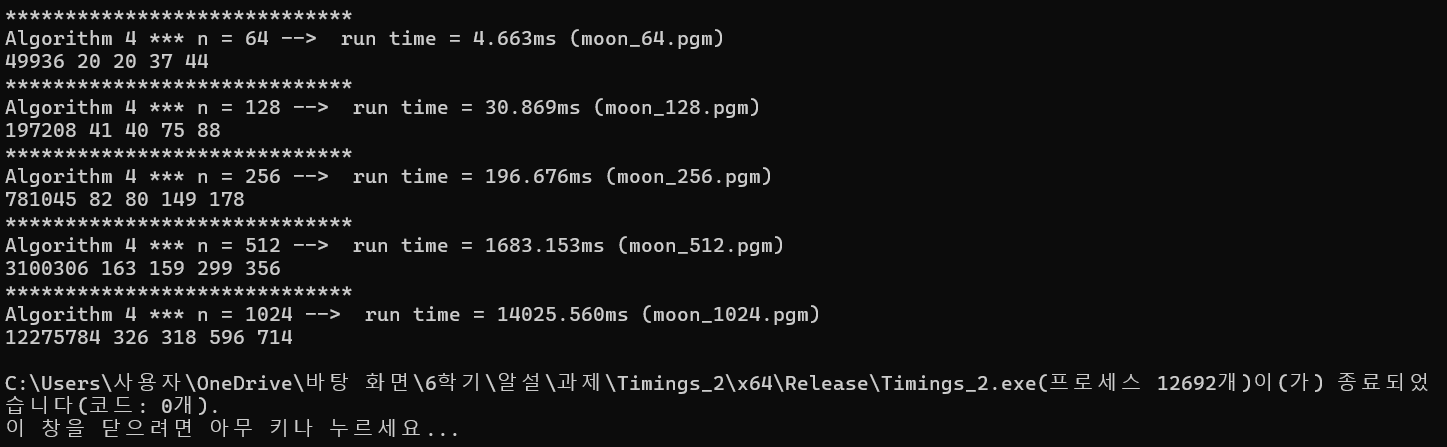
moon\_256.pgm의 평균 : (1509.66+1561.419+1534.282)/3=1535.12(ms)

moon\_512.pgm의 평균 : (24106.322+23819.080+23774.029)/3=23899.81(ms)

moon-Algorithm4







moon\_64.pgm의 평균 : (4.652+4.817+4.663)/3=4.711

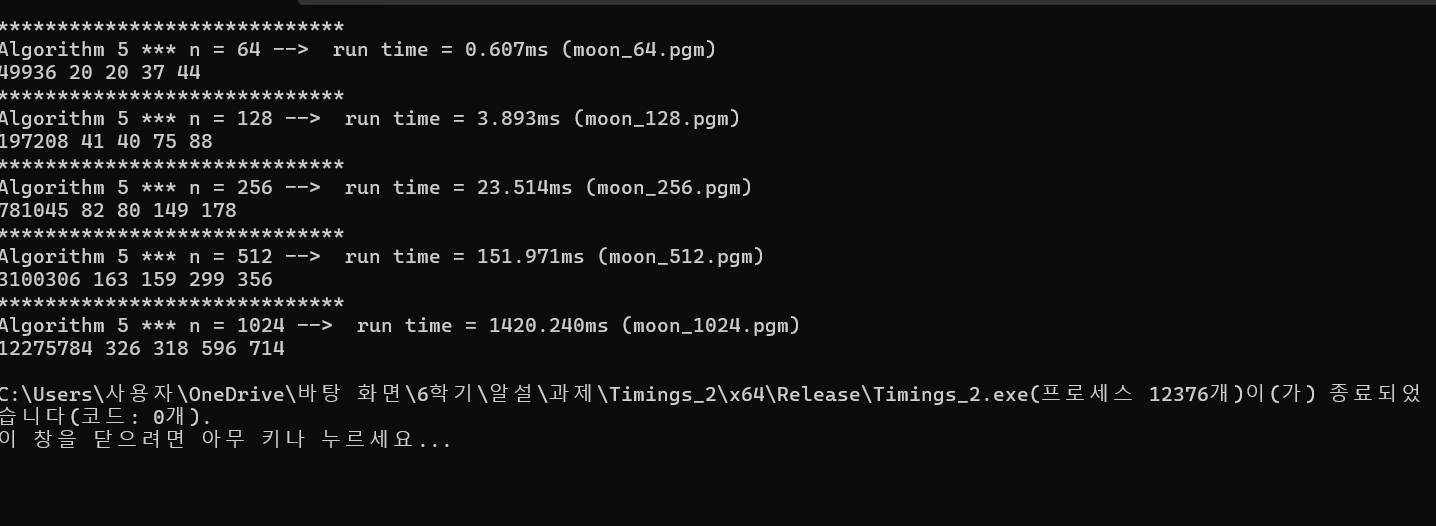
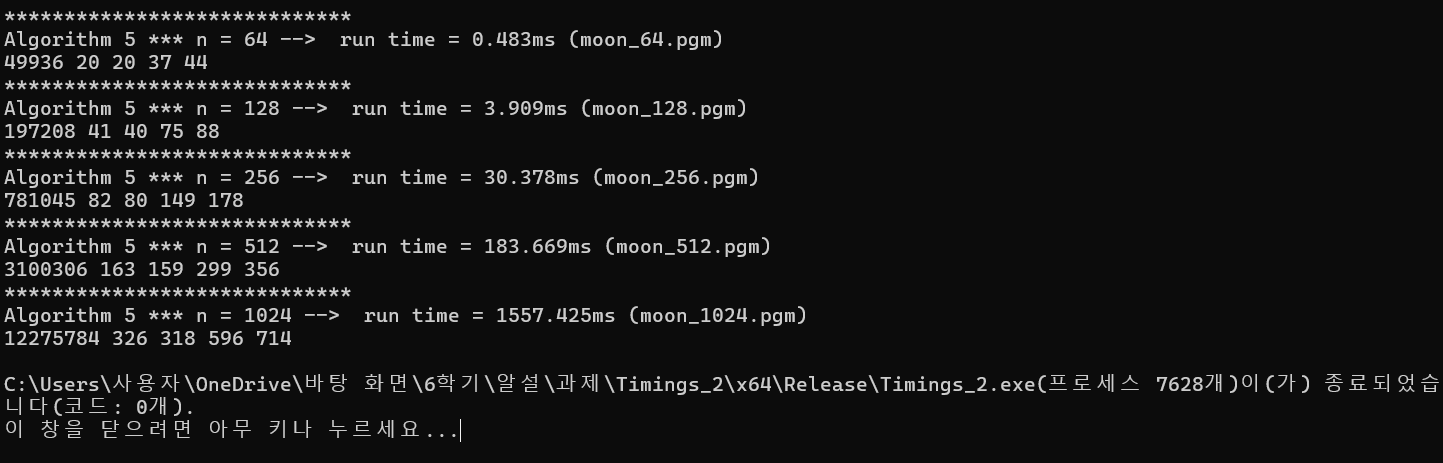
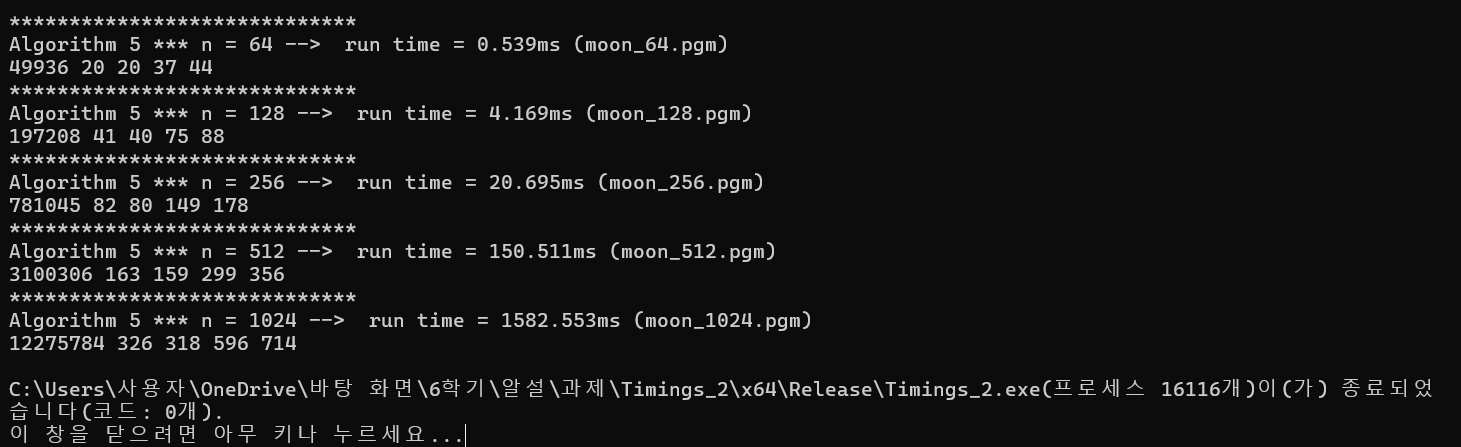
moon\_128.pgm의 평균 : (39.189+31.926+30.869)/3=33.99

moon\_256.pgm의 평균 : (210.824+202.920+196.676)/3=203.47

moon\_512.pgm의 평균 : (1667.377+1710.929+1683.153)/3=1687.153

moon\_1024.pgm의 평균 : (14068.725+14413.326+14025.560)/3=14169.20

moon-Algorithm5

moon\_64.pgm의 평균 : (0.607+0.483+0.539)/3=0.543

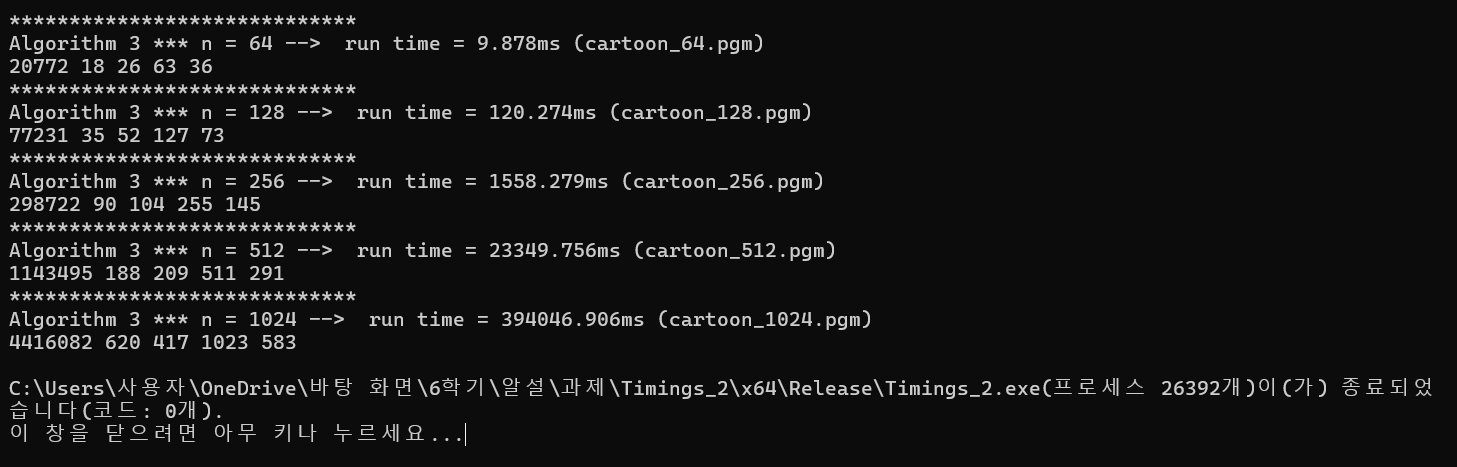
moon\_128.pgm의 평균 : (3.893+3.909+4.169)/3=3.99

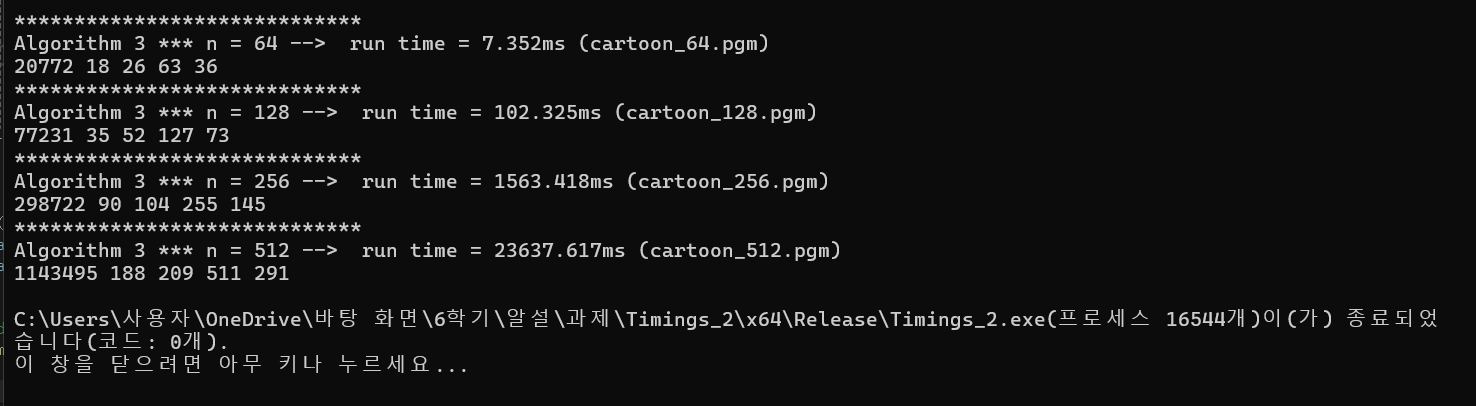
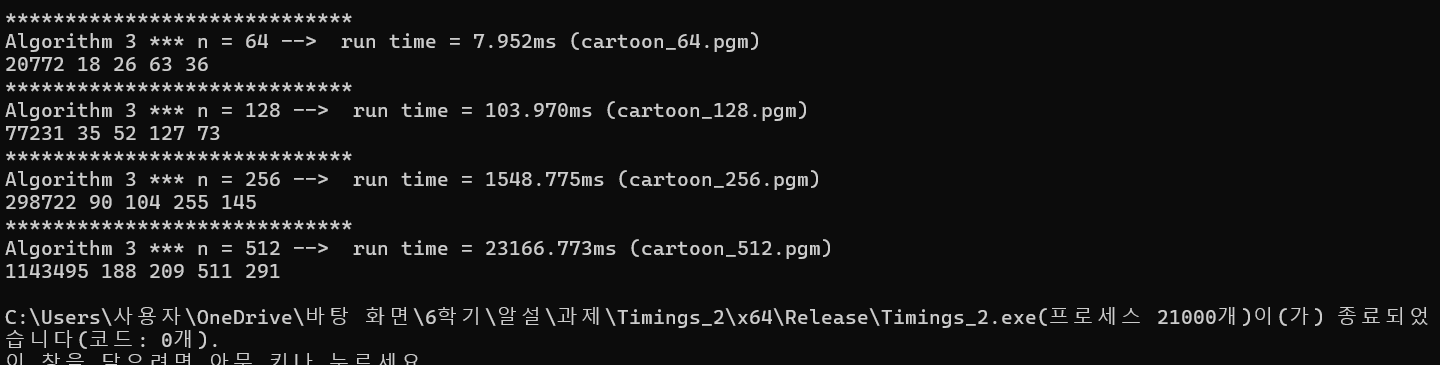
moon\_256.pgm의 평균 : (23.514+30.378+20.695)/3=24.86

moon\_512.pgm의 평균 : (151.971+183.669+150.511)/3=162.05

moon\_1024.pgm의 평균 : (1420.24+1557.425+1582.553)/3=1520.07

cartoon-Algorithm3



1024X1024는 너무 오래걸려서 실험을 한 번만 진행하였습니다.

cartoon\_64.pgm의 평균 : (9.878+7.352+7.952)/3=8.394

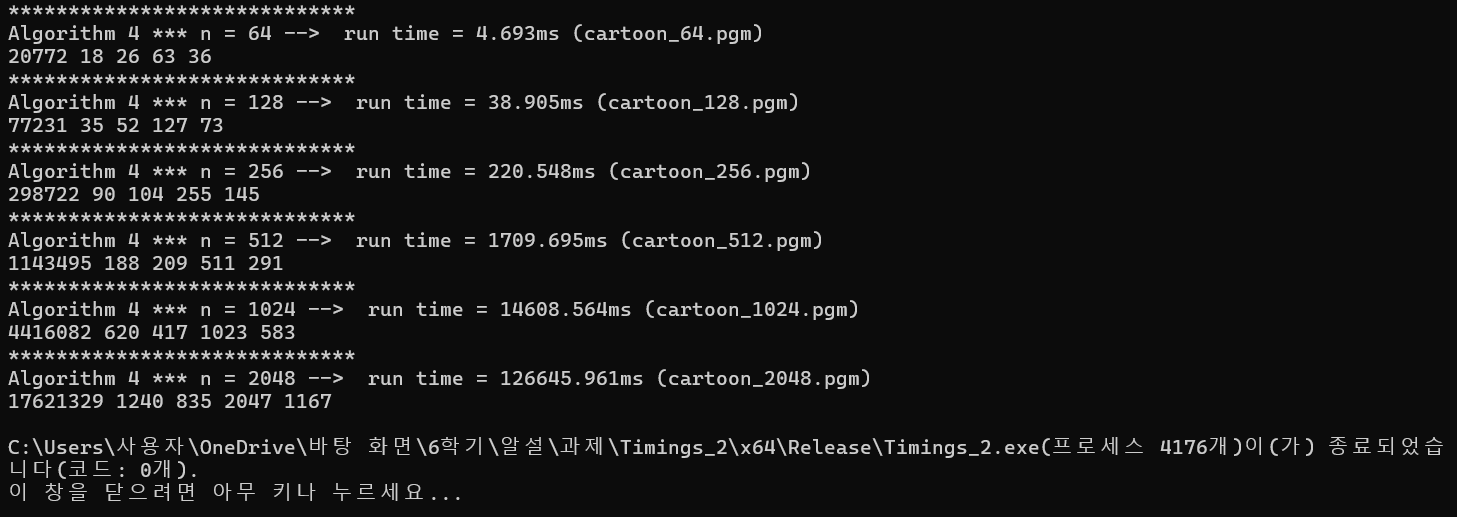
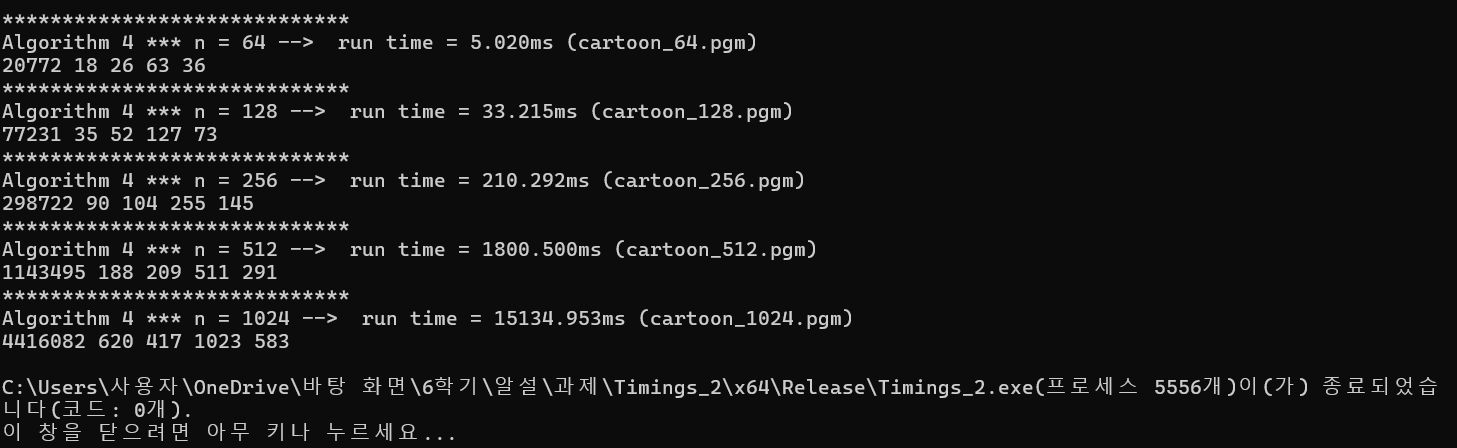
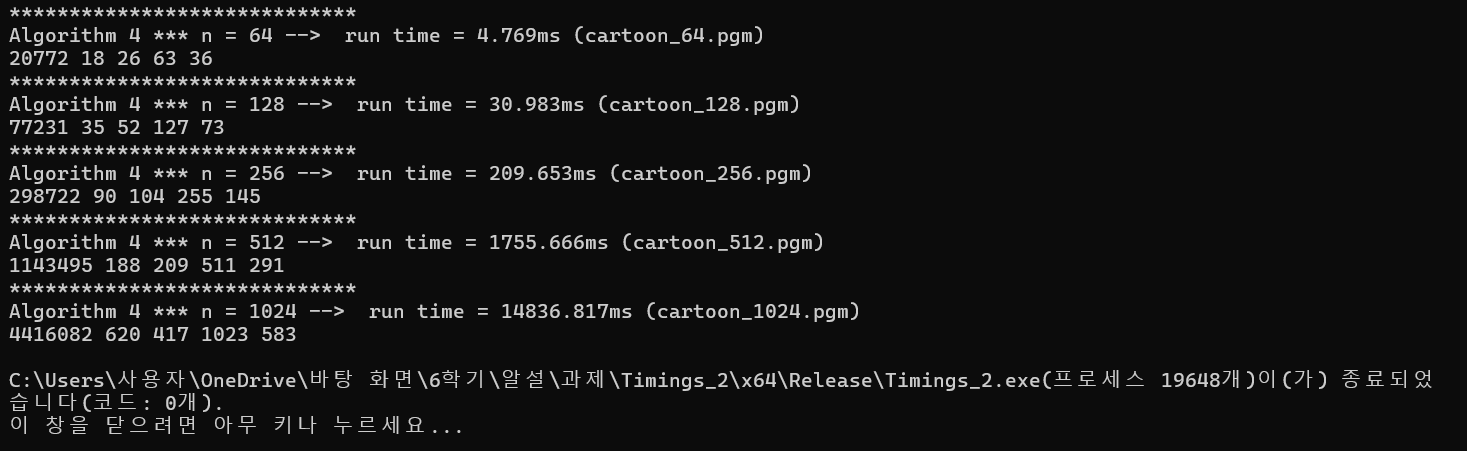
cartoon\_128.pgm의 평균 : (120.274+102.325+103.970)/3=108.86

cartoon\_256.pgm의 평균 : (1558.279+1563.418+1548.775)/3=1556.824

cartoon\_512.pgm의 평균 : (23349.756+23637.617+23166.773)/3=23384.72

cartoon\_1024.pgm : 394046.91

cartoon-Algorithm4

2048X2048은 오래걸려서 실험을 한 번만 진행하였습니다.

cartoon\_64.pgm의 평균 : (4.693+5.020+4.769)/3 = 4.83

cartoon\_128.pgm의 평균 : (38.905+33.215+30.983)/3 = 34.37

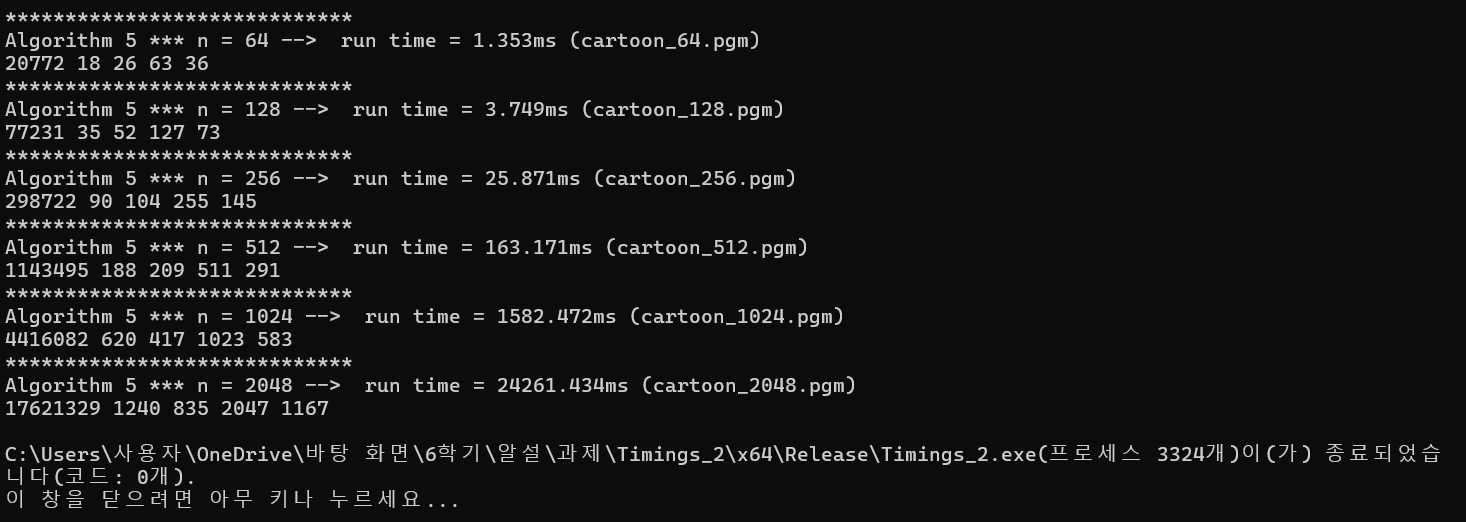
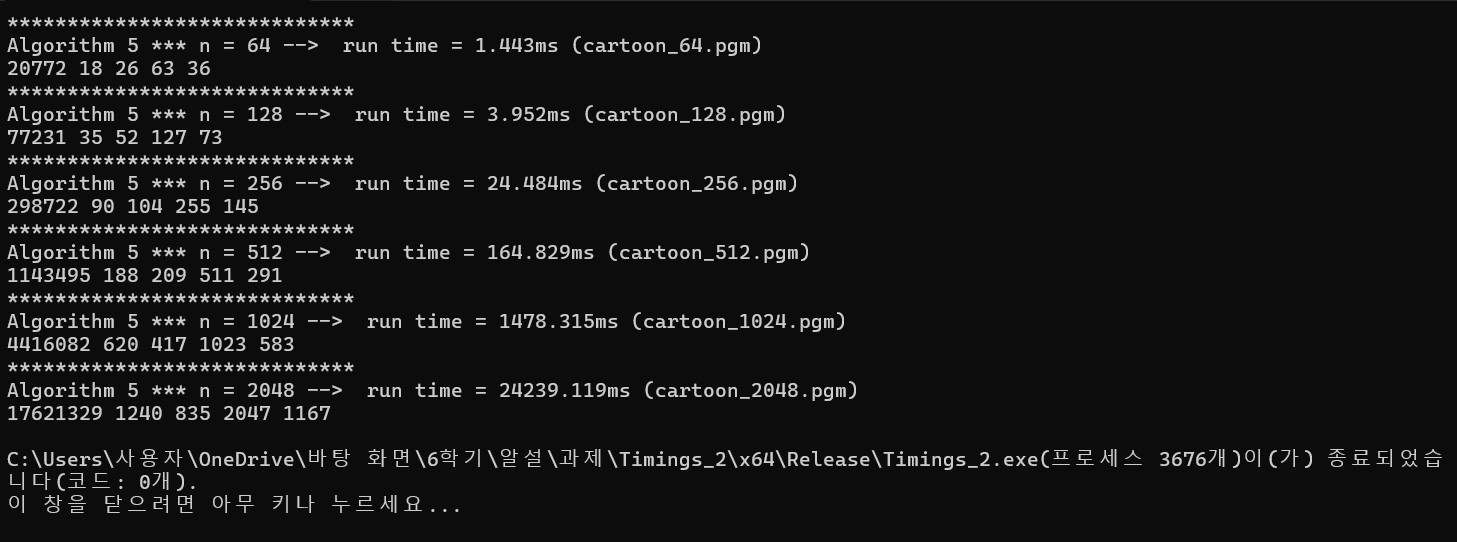
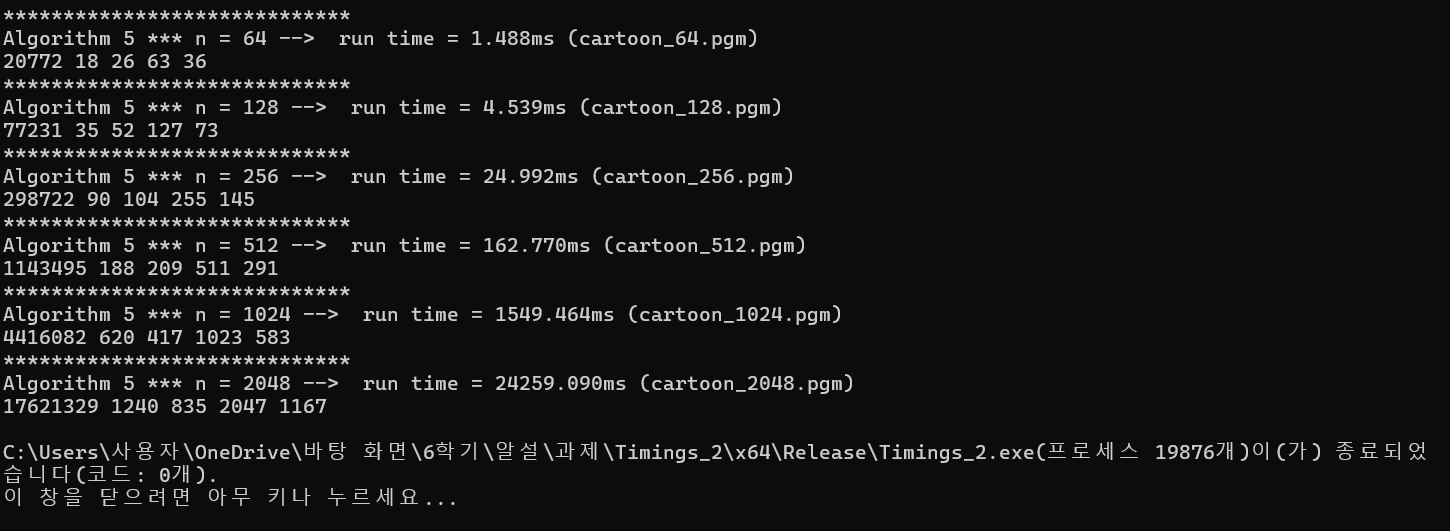
cartoon\_256.pgm의 평균 : (220.548+210.292+209.653)/3 = 213.50

cartoon\_512.pgm의 평균 : (1709.695+1800.500+1755.666)/3 = 1755.29

cartoon\_1024.pgm의 평균 : (14608.564+15134.953+14836.817)/3 = 14806.11

cartoon\_2048.pgm : 126645.961

cartoon-Algorithm5

cartoon\_64.pgm의 평균 : (1.353+1.443+1.488)/3 = 1.428

cartoon\_128.pgm의 평균 : (3.749+3.952+4.539)/3 = 4.08

cartoon\_256.pgm의 평균 : (25.871+24.484+24.992)/3 = 25.12

cartoon\_512.pgm의 평균 : (163.171+164.829+162.770)/3 = 163.59

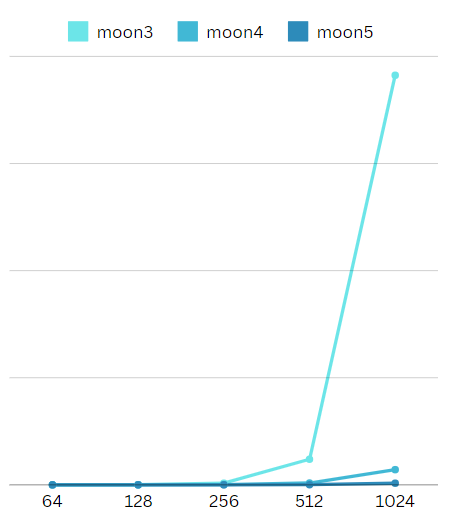
cartoon\_1024.pgm의 평균 : (1582.472+1478.315+1549.464)/3 = 1536.75

cartoon\_2048.pgm의 평균 : (24261.434+24239.119+24259.090)/3 = 24253.21

1.Time-Table(단위는 ms)

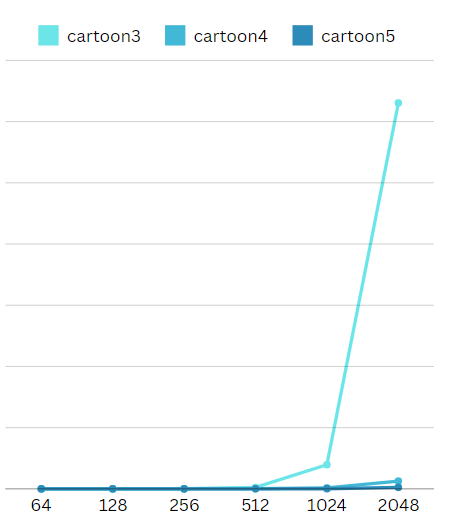
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | moon 3 | moon 4 | moon 5 | cartoon 3 | cartoon 4 | cartoon 5 |
| 64 | 9.64 | 4.711 | 0.543 | 8.394 | 4.83 | 1.428 |
| 128 | 147.22 | 33.99 | 3.99 | 108.86 | 34.37 | 4.08 |
| 256 | 1535.12 | 203.47 | 24.86 | 1556.824 | 213.50 | 25.12 |
| 512 | 23899.81 | 1687.153 | 162.05 | 23384.72 | 1755.29 | 163.59 |
| 1024 | X | 14169.20 | 1520.07 | 394046.91 | 14806.11 | 1536.75 |
| 2048 | X | X | X | X | 126645.961 | 24253.21 |

2.Moon\_n.pgm



가로축은 moon\_n.pgm의 가로축입니다. 세로축은 실행 시간입니다.

3.cartoon\_n.pgm



가로축은 cartoon\_n.pgm의 가로축입니다. 세로축은 실행 시간입니다.

이론적으로 Algorithm 3는 Time Complexity가 O(n^4), Algorithm 4는 O(n^3\*logn), Algorithm 5는 O(n^3)이 나옵니다. 그렇기 때문에 NxN에서 N이 2배로 증가할 때, 이상적으로 Algorithm 3은 2^4=16배, Algorithm 5는 2^3=8배가 증가합니다. Alogrithm 4의 이상적인 값은 log n을 하면서 Time Complexity 만으로는 예측을 할 수 없었습니다. 다만 Algrithm 3과 5사이의 비율을 가지지 않을까 예측해 보았습니다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | moon3 | moon4 | moon5 | cartoon3 | cartoon4 | cartoon5 |
| 64->128 | 15.3 | 7.2 | 7.34 | 12.97 | 7.115 | 2.85 |
| 128->256 | 10.4 | 5.98 | 6.23 | 14.3 | 6.218 | 6.16 |
| 256->512 | 15.57 | 8.29 | 6.518 | 15.0 | 8.22 | 6.5 |
| 512->1024 | X | 8.39 | 9.38 | 16.85 | 8.44 | 9.39 |
| 1024->2048 | X | X | X | X |  | 15.78 |

위 그래프는 nXn에서 2nX2n으로 변화할 때 실제 시행시간의 비율입니다.

moon3의 변화 비율의 평균은 13.53배가 나왔으며, moon4의 변화 비율의 평균은 7.39배가 나왔으며, moon5의 변화 비율의 평균은 7.27배가 나왔습니다.

cartoon3의 변화 비율의 평균은 14.71배가 나왔으며, cartoon4의 변화 비율의 평균은 7.44배가 나왔으며, 7.00배가 나왔습니다.

Algorithm 3은 13.53,14.71배로 16배보보다는 작지만 비슷한 결과를 얻었습니다.

Algorithm 5는 7.27,7.00배로 8배보다는 작지만 비슷한 결과를 얻었습니다.

Algorithm 4는 Algorithm 3와 Algorithm 5의 사이 값이 나왔으나, Algorithm 3보다 Algorithm 5과 가까운 값이 측정되었습니다. 이는 log n이 n이 커질수록 n보다는 1과 더 가깝기 때문입니다.

다음은 제 컴퓨터 실험 환경입니다.

OS : Windows 11 Home

CPU : Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz

RAM : 8.00GB

Compiler : Visual Studio 22 Release Mode/x64 Platform