알고리즘 설계와 분석

(Design and Analysis of Algorithms)

Machine Problem 1

20161603
신민준

Table of Contents

1 Implementations

1.1 $O(n^6)$ implementation

```c
struct _position {
   int row;
   int col;
};
typedef struct _position* position;

/**
 * Function to get maximum sum of subrectangle
 * with time complexity of O(n^6)
 * @param  matrix Pointer to matrix containing the data
 * @param  m      Number of rows in the matrix
 * @param  n      Number of columns in the matrix
 * @return        Maximum sum value of subrectangle
 */
int mss_pow6(int** matrix, int m, int n) {
   int mss = MIN_INT, sum;
   position from, to;

   from = malloc(sizeof(struct _position));
   to = malloc(sizeof(struct _position));

   from->row = from->col = 0;
   to->row = to->col = 0;

   // 6 loops. wow. seriously?
   for(from->row = 0 ; from->row < m ; from->row++) {
      for(from->col = 0 ; from->col < n ; from->col++) {
         for(to->row = from->row ; to->row < m ; to->row++) {
            for(to->col = from->col ; to->col < n ; to->col++) {
               sum = 0;
               for(int i=from->row ; i<=to->row ; i++) {
                  for(int j=from->col ; j<=to->col ; j++) {
                     sum += matrix[i][j];
                  }
               }
               if(sum > mss) mss = sum;
            }
         }
      }
   }

   free(from);
   free(to);
   return mss;
}
```

Code 1

가장 naive한 방법의 구현으로, 가능한 모든 직사각형을 구해 그 합이 최소가 되는 값을 찾는 구현 방식입니다.

가능한 모든 직사각형을 구하기 위해 좌표를 나타내는 position 구조체를 선언했고, 직사각형의 좌상단과 우하단 좌표를 정하기 위한 4중 for loop을 돌면서 동시에 해당 직사각형의 모

든 값을 더하는 과정을 수행하는 데 2중 for loop을 돌게 됩니다.

따라서, matrix의 크기가 $n \times n$이라고 할 때, 이 구현방식의 Time complexity는 $O(n^6)$입니다.

1.2 $O(n^4)$ implementation

```
/**
 * Function to get maximum sum of subrectangle
 * with time complexity of O(n^4)
 * @param  matrix Pointer to matrix containing the data
 * @param  m      Number of rows in the matrix
 * @param  n      Number of columns in the matrix
 * @return        Maximum sum value of subrectangle
 */
int mss_pow4(int** matrix, int m, int n) {
  int mss = MIN_INT, partial_mss = MIN_INT;
  int* sub_arr = malloc(sizeof(int)*m);

  for(int left = 0 ; left < n ; left++) {
    memset(sub_arr, 0, m*sizeof(int));
    for(int right = left ; right < n ; right++) {
      for(int i=0 ; i<m ; i++)
        sub_arr[i] += matrix[i][right];

      int arr_sum;
      for(int i=0 ; i<m ; i++) {
        arr_sum = 0;
        for(int j=i ; j<m ; j++) {
          arr_sum += sub_arr[j];
          if(arr_sum > mss)
            mss = arr_sum;
        }
      }
    }
  }
  free(sub_arr);
  return mss;
}
```

Code 2

2D 배열에서 '왼쪽 열'과 '오른쪽 열'을 설정하고, 두 열 사이에서 inclusive하게 만들어지는 2차원 배열의 row element들을 각각 더해 1차원 배열로 바꾼 후, 이 1차원 배열에 대해 $O(n^2)$의 Time complexity를 가지는 Maximum Sum Subsequence 알고리즘을 사용해 최댓값을 구하는 알고리즘입니다. 왼쪽 열과 오른쪽 열을 정하는 과정에서 2개의 for loop을 사용하였고, Maximum Sum Subsequence를 구하는 알고리즘에서 2개의 for loop이 사용되어, 알고리즘의 전체 Time complexity는 다음과 같습니다.

$$O(n^4), when\ matrix's\ size\ is\ n \times n$$

1.3 $O(n^3)$ implementation

```
/**
 * Kandane's algorithm to calculate Maximum Subsequence Sum
 * @param  arr    Array containing the data.
 * @param  length Length of the array
 * @return        Value of MSS.
 */
int mss_subseq(int* arr, int length) {
   int sum, mss = MIN_INT;
   sum = 0;
   for(int i=0 ; i<length ; i++) {
      sum += arr[i];
      if(sum > mss) mss = sum;
      if(sum < 0) sum = 0;
   }
   return mss;
}
```

Code 3

1.2에서 소개한 알고리즘에서 Maximum Sum Subsequence를 구하는 부분의 알고리즘을 위 [Code 3]에서 볼 수 있는 Kandane's algorithm으로 대체했습니다. 이를 사용해 구현한 알고리즘과 Time complexity는 다음과 같습니다.

```
/**
 * Function to get maximum sum of subrectangle using Kandane's Algorithm.
 * with time complexity of O(n^3)
 * @param  matrix Pointer to matrix containing the data
 * @param  m      Number of rows in the matrix
 * @param  n      Number of columns in the matrix
 * @return        Maximum sum value of subrectangle
 */
int mss_pow3(int** matrix, int m, int n) {
   int mss = MIN_INT, partial_mss = MIN_INT;
   int* sub_arr = malloc(sizeof(int)*m);

   for(int left = 0 ; left < n ; left++) {
      memset(sub_arr, 0, m*sizeof(int));
      for(int right = left ; right < n ; right++) {
         for(int i=0 ; i<m ; i++)
            sub_arr[i] += matrix[i][right]; // create 1-D array from matrix
         partial_mss = mss_subseq(sub_arr, m);
         if(partial_mss > mss)
            mss = partial_mss;
      }
   }
   free(sub_arr);
   return mss;
}
```

Code 4

$$O(n^3), when\ matrix\ is\ of\ size\ n \times n$$

2 Experiment

2.1 Environment Specifications

MacBook Pro (13-inch, 2018)
OS - macOS Mojave version 10.14.6
CPU - 2.3 GHz Intel Core i5
RAM - 8GB 2133 MHz LPDDR3
Graphics - Intel Iris Plus Graphics 655 1538MB
Shell - zsh 5.7.1 (x86_64-apple-darwin18.2.0)
Compiler - Apple LLVM version 10.0.1 (clang-1001.0.46.4)

2.2 Experiment Setup

주어진 make_input 프로그램을 사용해 실험에 필요한 input 파일들을 생성했습니다. 생성할 때 사용한 argument들은 다음과 같습니다.

```
: 1570083090:0;./make_input 1.txt 10 10 0 100 S33D
: 1570083098:0;./make_input 2.txt 30 30 0 100 S33D
: 1570083104:0;./make_input 3.txt 50 50 0 100 S33D
: 1570083109:0;./make_input 4.txt 70 70 0 100 S33D
: 1570083120:0;./make_input 4-1.txt 70 70 -100 -1 S33D
: 1570083133:0;./make_input 3-2.txt 50 50 -100 -1 S33D
: 1570083143:0;./make_input 5.txt 100 100 0 100 S33D
```

Figure 1 ~/.zsh_history

4-1.txt와 3-2.txt는 행렬을 구성하는 값이 모두 음수일 경우의 Time complexity 차이를 알아보기 위해 추가로 생성했습니다.

2.3 Result Data

모든 값들의 단위는 ms입니다.

n	10	30	50	70	100
Algorithm 1	0.216000	59.088000	973.343000	6878.017000	55427.870000
Algorithm 2	0.027000	0.712000	4.910000	17.275000	64.602000
Algorithm 3	0.013000	0.099000	0.475000	1.225000	2.719000

Table 1

n	50	50(neg. valued)	70	70(neg. valued)
Algorithm 1	973.343000	974.673000	6878.017000	6847.705000
Algorithm 2	4.910000	4.595000	17.275000	16.787000
Algorithm 3	0.475000	0.469000	1.225000	1.056000

Table 2

2.4 Data Analysis

[Table 1]의 자료를 그래프로 표현했을 때 다음과 같습니다.

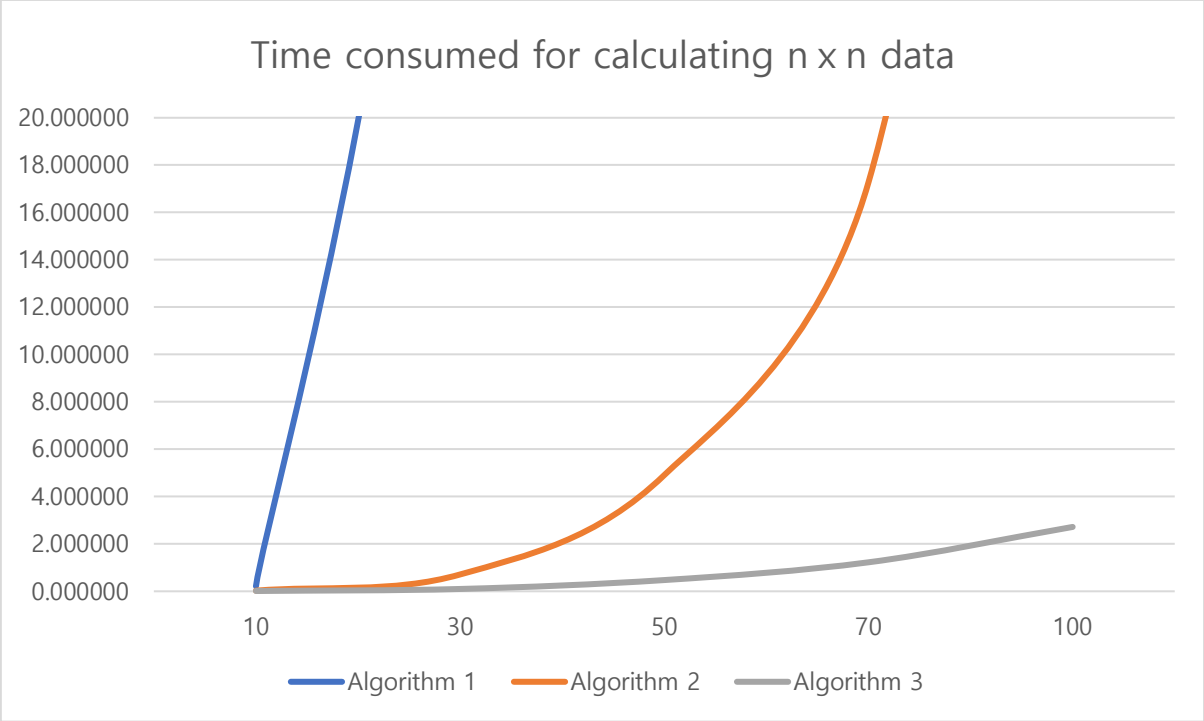

Figure 2

X축: n의 값, Y축: execution time in milliseconds

[Figure 2]의 자료의 Y-value를 최대 20ms까지만 표시한 이유는, Algorithm 1의 증가폭이 너무나도 커 다른 두 알고리즘과의 상관관계를 명확하게 보이기 힘들었기 때문입니다. [Figure 2]의 Y 축에 로그를 사용해 새롭게 그래프를 그리면 다음 [Figure 3]와 같습니다.

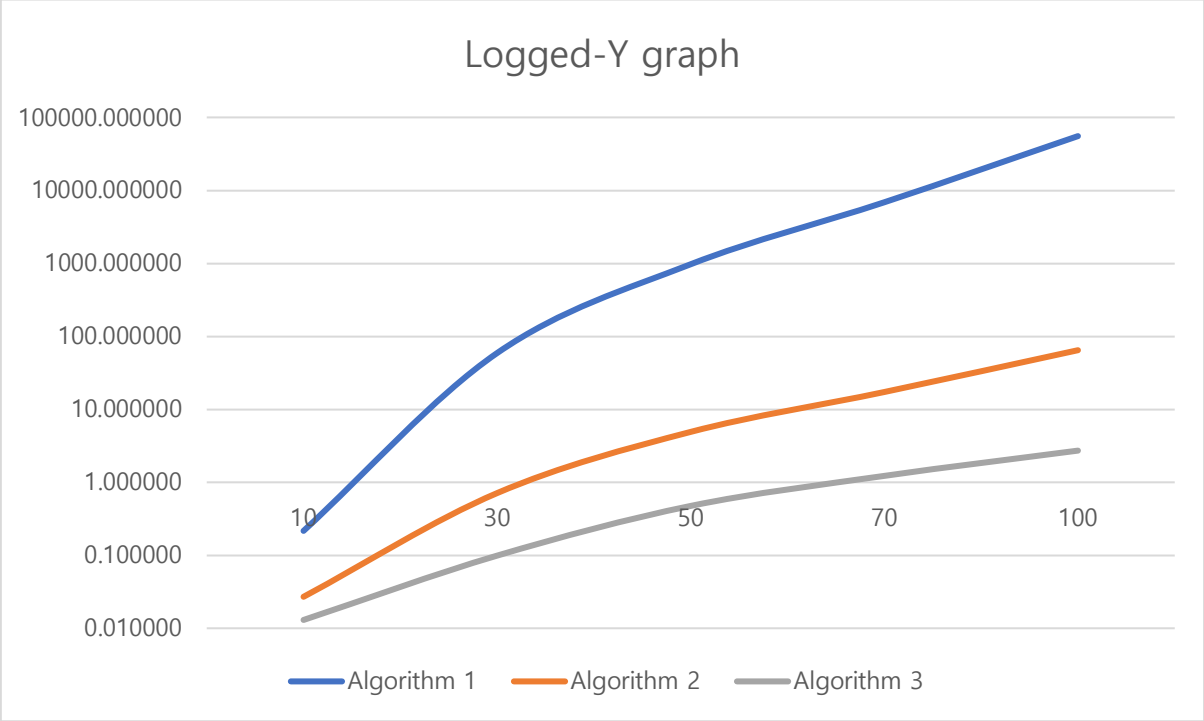

Figure 3

X축: n의 값, Y축: 로그 시간

[Figure 2]와 [Figure 3]에서 확연하게 드러나는 것은, 세 알고리즘의 증가폭이 서로 크게 차이가 난다는 점입니다. 이는 각 알고리즘이 $O(n^6), O(n^4), O(n^3)$의 Time complexity를 가진다는 점을 고려한다면 충분히 자연스러운 결과라 할 수 있습니다.

[Table 2]에서는 모든 값이 음수인 데이터와 모든 값이 양수인 데이터 사이의 처리 시간 차이를 알아보았는데, 결과적으로 데이터 속의 값이 무엇이든 상관없이 실행 시간은 n에 따라서만 변동한 것을 확인할 수 있습니다.

3 Additional Comments

- 1 차원 배열의 MSS 문제해결에서 낮은 Time complexity 를 가지는 Kandane 알고리즘을 사용하고 싶었고, 이를 위해 주어진 2 차원 배열을 1 차원 배열로 적절하게 변환했습니다. 작은 차원에서의 해결 방법을 사용해 더 높은 차원에서의 문제를 해결하는 bottom-up 방식의 문제 해결 방법이라 볼 수 있습니다.
- Non-positive 값만 가지는 행렬의 경우, 모든 $n \times n$개의 element 중 가장 큰 값을 가진 element 가 Maximum Sum Subrectangle Value 이므로, 구현한 알고리즘을 수행하기 전 주어진 행렬이 non-positive 값으로만 이루어져 있는지 확인하는 부분을 추가한다면,

전체 프로그램의 Time complexity 에는 변화가 없으면서 평균 수행시간을 휴리스틱하게 어느 정도 줄일 수 있을 것이라 기대할 수 있습니다.