Maximum Sum Subrectangle in a 2D array

과목 : 알고리즘 설계 및 분석

분반 : 2 반

교수님 : 소정민 교수님

학번 : 20141284

이름 : 이기현

1. 프로그램 실행 환경

* OS : Linux version 4.4.0-164-generic (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.10))
* CPU Model : Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz
* CPU MHz : 1632.1215
* Cache size : 25600 KB
* RAM : 65862640KB (= 65GB)

1. 함수 설명

int O\_6\_algorithm(vector<vector<int>> &num, int n, int m)

* 알고리즘 : Brute Force
* Input : num – 2차원 배열, n – row size, m – column size
* Output : 가장 큰 부분행렬의 합
* Function :
  + 행렬 내부에서 부분행렬의 위치를 잡고 모든 항목을 더해서 부분행렬의 합을 구한다. 이렇게 구한 부분행렬의 합을 모든 부분행렬의 합과 비교하여 최대값을 구한다.
  + 부분행렬의 좌측 상단을 2중 for문으로 구하기O(nm))
  + 부분행렬의 우측 하단을 2중 for문으로 구하기(O(nm))
  + 위에서 구한 좌측 상단 ~ 우측 하단의 부분행렬의 합을 2중 for문으로 구하기 (O(nm))

int O\_4\_algorithm(vector<vector<int>> &num, int n, int m)

* 알고리즘 : Dynamic Programming
* Input : num – 2차원 배열, n – row size, m – column size
* Output : 가장 큰 부분행렬의 합
* Function :
  + Maximum Subsequence Sum에서 O(n) 의 시간복잡도로 해결 했던 것을 잡기 위해, 행렬의 i~j row를 압축한 뒤, 압축된 m-size의 sequence에 대해 maximum subsequence sum 을 구하고 모든 (i, j) 쌍과 비교하여 최대값을 구한다.
  + (I, j) 쌍 구하기 O(n^2)
  + 압축된 m-size의 sequence를 읽기 O(m)
  + Sequence를 읽으면서 압축되는 값을 찾기 O(n)

int O\_3\_algorithm(vector<vector<int>> &num, int n, int m)

* 알고리즘 : Prefix Sum & Dynamic Programming
* Input : num – 2차원 배열, n – row size, m – column size
* Output : 가장 큰 부분행렬의 합
* Function :
  + O(4) 알고리즘과 비슷한 방식이지만, i~j row를 압축하는 과정에서 매번 압축하는 것이 아닌 행렬의 column별로 prefix sum을 저장하여 압축된 부분의 합을 O(1)에 구할 수 있도록 한다.
  + (I, j) 쌍 구하기 O(n^2)
  + 압축된 m-size의 sequence를 읽기 O(m)
  + Sequence를 읽으면서 압축되는 값을 찾기 O(1)

1. Input 사이즈에 따른 함수 별 수행시간 비교(n = m case)

* Input data 설명
  + Size : 5\*5, 10\*10, 15\*15, 25\*25, 40\*40, 55\*55, 80\*80, 100\*100
  + Max value : 1000
  + Min value : -1000
  + Seed value : 141284

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 5 \* 5 | 10 \* 10 | 15 \* 15 | 25 \* 25 | 40 \* 40 | 55 \* 55 | 80 \* 80 | 100 \* 100 |
| O(n^6) | 0.030 | 0.867 | 7.887 | 109.724 | 1047.9 | 5546.35 | 50386.6 | 183335 |
| O(n^4) | 0.014 | 0.067 | 0.254 | 1.502 | 8.374 | 27.888 | 96.072 | 187.279 |
| O(n^3) | 0.017 | 0.034 | 0.083 | 0.531 | 2.201 | 5.291 | 16.026 | 25.627 |

* 수행시간 비교 (단위 : ms)

(모든 수행은 7~8 회 실행해서 중간값으로 판단되는 시간을 기록하였습니다.)

(Comments 뒷부분에 위 표에 대한 그래프를 첨부하였습니다.)

* Comments

가장 작은 사이즈인 5 \* 5 파일을 테스트했을 때는 O(n^3)알고리즘의 수행시간이 상대적으로 크게 나왔음을 확인할 수 있습니다. O(n^3)은 함수 내부에서 n\*m 사이즈의 벡터 array와 m사이즈의 벡터 array를 생성하고 n\*m 사이즈의 array에 prefix\_sum을 생성해주는 과정이 있습니다. 그렇기 때문에 이에 따른 시간이 전체 수행시간에 영향을 미쳤다고 짐작할 수 있습니다. 또한 prefix sum을 생성하는 과정에서 2차원 배열에서의 for문을 통한 인덱스 접근이 row major 방식이 아닌 column major 방식으로 이루어지기 때문에 그에 따른 수행시간이 크게 작용하는 부분도 있습니다.

또, 10 \* 10 사이즈 이상의 input에 대한 테스트에서는 n^3 알고리즘에서 n^4 알고리즘으로 갈 땐, 1 ~ 8 배로 커지고 n^4 알고리즘에서 n^6 알고리즘으로 갈 땐, 2 ~1000 배정도 수행시간이 늘어난 것을 확인할 수 있었습니다.

그래프1. O(n^4), O(n^3) 알고리즘의 Input size에 따른 수행시간

그래프2. O(n^6) 알고리즘의 Input size에 따른 수행시간

1. Input 사이즈에 따른 함수 별 수행시간 비교(n /= m case)

* Input data 설명
  + Size : 10\*50, 50\*10, 20\*100, 100\*20, 40\*50, 50\*40, 25\*300, 300\*25
  + Max value : 1000
  + Min value : -1000
  + Seed value : 141284

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 \* 50 | 50 \* 10 | 20 \* 100 | 100 \* 20 | 40 \* 50 | 50 \* 40 | 25 \* 300 | 300 \* 25 |
| O(n^3m^3) | 67.021 | 60.958 | 1919.63 | 1937.17 | 1741.48 | 1884.39 | 83197.9 | 86722.4 |
| O(n^3m^1) | 0.27 | 4.167 | 2.711 | 51.446 | 10.438 | 15.618 | 14.602 | 811.252 |
| O(n^2m^1) | 0.121 | 0.86 | 1.376 | 6.541 | 2.678 | 1.789 | 5.946 | 60.632 |

* 수행시간 비교 (단위 : ms)
* Comments

위 표에 작성하였 듯이 n과 m이 다른 경우 시간복잡도가 n과 m으로 구성된다. 이 때, n과 m중 어떤 값이 전체 시간에 더 영향을 미치는지 확인할 수 있습니다.

O(n^3m^3) = O(n^6) 알고리즘의 경우엔 n과 m의 차수가 같으므로 영향력이 비슷하다. 그 결과로 n과 m을 바꿔 수행했을 때 수행시간이 비슷한 것을 확인할 수 있었습니다.

O(n^3m^1) = O(n^4) 과 O(n^2m^1) = O(n^3) 알고리즘 경우에는 n의 차수가 더 크기 때문에 그만큼 영향력이 강한 것을 확인할 수 있습니다. 그 결과로 10 \* 50 Input보다는 50 \* 10 Input에서, 마찬가지로 n과 m값만 바꾼 모든 케이스에서 n이 m보다 클 때, 시간이 적게는 2~3배 많게는 5~600배까지 차이나는 것을 확인할 수 있습니다.

또한, 20 \* 100(100 \* 20) Input과 40 \* 50(50 \* 40) Input의 전체 셀의 개수는 2000개로 동일하지만 후자의 Input에서 n과 m의 사이즈가 크게 차이가 나지 않아 n이 더 작은 경우엔 m의 영향력이 조금 더 커져 전자보다 수행시간이 길었고, n이 더 큰 경우엔 n의 영향력이 n의 영향력이 약해져 전자보다 수행시간이 짧았습니다.