**알고리즘 설계와 분석**

**HW1 보고서**

**20140938**

**임다은**

**HW A. Maximum Subsequence Sum Problem**

**1. 실험 환경**

OS: macOS Sierra

CPU: 2.7GHz Intel Core i5에서 진행하였다.

RAM: 8GB 1867 MHz DDR3이다.

Compiler: g++ -std=c++11

**2. 입력 데이터**

**a) 데이터의 정확성 향상을 위한 실험 방식**

n 값은 2^20에 가까운 1,000,000까지 값을 주었다. 가장 작은 값도 수행결과를 측정하는 데 의미가 있도록 충분히 큰 값을 구하기 위하여 10,000으로 하였다. 또한 가장 작은 값부터 가장 큰 값까지 n의 분포가 충분히 흩어져 있도록 설정하였다.

이에 따라 각 값으로 10,000, 50,000, 100,000, 500,000, 1,000,000을 가지는 데이터를 만들었다. 이 때 하나의 n 값에 대하여 다양한 데이터를 바탕으로 실험하기 위해 5개의 다른 실험데이터를 만들었다. 또한 하나의 데이터를 가지고 수행하는 평균 값을 구하기 위해 데이터당5번의 실험을 진행하여 평균값을 구하였다.

**b) HW1\_MSS\_config.txt 구성**

인풋 파일은 'MSS\_dd.input'으로 구성된다. 이 때 첫번째 d는 총 다섯 가지의 n의 크기를 0부터 4까지로 대체한 수이다. 이 때 각 n에 대해 5개의 데이터가 존재하므로 두번째 d는 이를 나타내고 있다.

아웃풋 파일은 MSS\_dd\_dd.output.txt로 구성된다. 앞의 두개의 d는 인풋과 같은 숫자를 가진다. 뒤의 d 중 첫번째 숫자는 세가지 함수 중 사용한 함수의 숫자를, 마지막 d는 한 데이터 당 다섯번의 실험을 하는 것을 나타내고 있다.

**3. 결과 데이터**

a) 결과 데이터

1) Input size: 10,000

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Data** | **A1** | **A2** | **A3** |
| MSS\_00.input | 135.726 | 1.671 | 0.042 |
| MSS\_01.input | 132.579 | 1.687 | 0.032 |
| MSS\_02.input | 132.66 | 2.021 | 0.059 |
| MSS\_03.input | 143.192 | 2.301 | 0.048 |
| MSS\_04.input | 138.651 | 2.284 | 0.04 |
| Average | 136.562 | 1.993 | 0.044 |

<표1> input size 10,000일 때 함수 별 수행 시간

2) Input size: 50,000

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Data** | **A1** | **A2** | **A3** |
| MSS\_10.input | 4,150.719 | 15.451 | 0.186 |
| MSS\_11.input | 4,670.304 | 23.688 | 0.189 |
| MSS\_12.input | 3,901.318 | 10.093 | 0.142 |
| MSS\_13.input | 3,704.91 | 10.744 | 0.144 |
| MSS\_14.input | 3,645.783 | 11.782 | 0.181 |
| Average | 4,014.607 | 14.352 | 0.168 |

<표2> input size 50,000일 때 함수 별 수행 시간

3) Input size: 100,000

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Data** | **A1** | **A2** | **A3** |
| MSS\_20.input | 15,566.264 | 23.018 | 0.365 |
| MSS\_21.input | 13,443.223 | 23.174 | 0.263 |
| MSS\_22.input | 13,572.996 | 24.419 | 0.326 |
| MSS\_23.input | 13,381.465 | 32.759 | 0.808 |
| MSS\_24.input | 13,966.128 | 27.552 | 0.3 |
| Average | 13,986.015 | 26.184 | 0.412 |

<표3> input size 100,000일 때 함수 별 수행 시간

4) Input size: 500,000

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Data** | **A1** | **A2** | **A3** |
| MSS\_30.input | 338,346.122 | 127.741 | 4.602 |
| MSS\_31.input | 335,162.878 | 102.784 | 1.341 |
| MSS\_32.input | 320,459.227 | 104.135 | 2.517 |
| MSS\_33.input | 320,067.305 | 95.201 | 2.044 |
| MSS\_34.input | 328,533.844 | 99.718 | 1.316 |
| Average | 328,513.875 | 105.9158 | 2.364 |

<표4> input size 500,000일 때 함수 별 수행 시간

5) Input size: 1,000,000

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Data** | **A1** | **A2** | **A3** |
| MSS\_40.input | 1,258,895.831 | 184.904 | 2.721 |
| MSS\_41.input | 1,261,523.177 | 184.65 | 3.196 |
| MSS\_42.input | 1,260,916.161 | 195.967 | 4.259 |
| MSS\_43.input | 1,262,260.063 | 188.957 | 2.56 |
| MSS\_44.input | 1,268,575.244 | 178.552 | 2.541 |
| Average | 1,262,434.095 | 186.606 | 3.055 |

<표5> input size 1,000,000일 때 함수 별 수행 시간

**3. 데이터 분석 결과**

**a) 수행 결과의 그래프 및 수식**

그림1. 각 알고리즘 별 수행시간, Y축은 logarithmic하게 커진다.

**b) 이론적인 시간복잡도와 수행 시간 관계 분석**

실험 데이터 사이즈가 각10,000, 50,000, 100,000, 500,000, 1,000,000이므로

실험한 CPU의 클럭은 2.7GHz이므로 1 millisecond당 2,700,000회 계산을 수행한다고 볼 수 있다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| tasks to CPU | n^2 | nlogn | | n |
| 10,000 | 100,000,000.000 | 40,000.000 | | 10,000.000 |
| 50,000 | 2,500,000,000.000 | 234,948.500 | | 50,000.000 |
| 100,000 | 10,000,000,000.000 | 500,000.000 | | 100,000.000 |
| 500,000 | 250,000,000,000.000 | 2,849,485.002 | | 500,000.000 |
| 1,000,000 | 1,000,000,000,000.000 | 6,000,000.000 | | 1,000,000.000 |
| **<표6> cpu가 처리할 계산의 개수** | | | | |
| expected cpu time | n^2 | nlogn | | n |
| 10,000 | 37.037 | 0.015 | | 0.004 |
| 50,000 | 925.926 | 0.087 | | 0.019 |
| 100,000 | 3,703.704 | 0.185 | | 0.037 |
| 500,000 | 92,592.593 | 1.055 | | 0.185 |
| 1,000,000 | 370,370.370 | 2.222 | | 0.370 |
| **<표7> cpu가 예상되는 처리 시간 (tasks to cpu / tasks cpu can process in 1 millisec)** | | | | |
| real cpu time | n^2 | | nlogn | n |
| 10,000 | 136.562 | | 1.993 | 0.044 |
| 50,000 | 4,014.61 | | 14.352 | 0.168 |
| 100,000 | 13,986.02 | | 26.184 | 0.412 |
| 500,000 | 328,513.88 | | 105.9158 | 2.364 |
| 1,000,000 | 1,262,434.10 | | 186.606 | 3.055 |
| **<표8> cpu가 실제 처리하는 데 걸린 시간** | | |
| expected / real cpu time | n^2 | | nlogn | n |
| 10,000 | 3.687 | | 134.528 | 11.880 |
| 50,000 | 4.336 | | 164.931 | 9.072 |
| 100,000 | 3.776 | | 141.394 | 11.124 |
| 500,000 | 3.548 | | 100.360 | 12.766 |
| 1,000,000 | 3.409 | | 83.973 | 8.249 |
| **<표9> 처리 시간 / 처리 시간 예상 값** | | |

**<그림 2> 예상되는 처리 시간 그래프, x축은 인풋 y축은 시간에 로그 취한 것**

**<그림 3> 실제 처리 시간 그래프, x축은 인풋 y축은 시간에 로그 취한 것**

루프 내의 상수 계수에 의해서 expected cpu time과 real cpu time이 같은 값을 가지진 않지만, 시간의 경향성은 매우 유사하게 증가함을 볼 수 있다. 표 9를 보면 실제 처리된 시간을 예상되는 시간으로 나눈 값이 함수별로 어느정도 일정한 것을 볼 수 있다. 알고리즘 1의 경우 이 값이 3.4에서 4.3으로, 알고리즘 2의 경우 약간 편차가 크지만 120에서 앞뒤로 40정도 차이, 알고리즘 3의 경우 약 10배의 수준을 유지하고 있다. 알고리즘 2의 경우 편차가 큰 이유는 다른 알고리즘에 비해 루프를 돌릴 때 step들이 많아 곱해지는 상수가 굉장히 큼을 알 수 있다.

**4. 결론**

이에 따라서 실제 알고리즘 처리 시간은 예상되는 처리 시간과 같은 만큼의 경향성을 가지고 있음이 확인되었으며, 함수 내의 루프와 기타로 처리되는 스텝의 개수에 따라서 Big-Oh 연산을 통한 실제 처리시간 분석에서 큰 값의 상수 배가 곱해지는 것을 알 수 있었다.

**HW B. Inversion Counting Problem**

**1. 실험 환경**

HW A와 동일한 실험환경에서 진행하였다.

**2. 입력 데이터**

**a) 데이터의 정확성 향상을 위한 실험 방식**

HW A와 동일한 방식으로 데이터 정확성 향상을 도모하였다.

**b) HW1\_MSS\_config.txt 구성**

인풋 파일은 'IC\_dd.input'으로 구성된다. 이 때 첫번째 d는 총 다섯 가지의 n의 크기를 0부터 4까지로 대체한 수이다. 이 때 각 n에 대해 5개의 데이터가 존재하므로 두번째 d는 이를 나타내고 있다.

아웃풋 파일은 IC\_dd\_d.output.txt로 구성된다. 앞의 두개의 d는 인풋과 같은 숫자를 가진다. 마지막 d는 한 데이터 당 다섯번의 실험을 하는 것을 나타내고 있다.

**3. 알고리즘 설계 방식**

**a) O(nlogn) 시간복잡도 구현 방법**

알고리즘 설계의 기본 방식은 Merge Sort와 같은 형식을 갖는다. O(nlogn) 구현은 divide and conquer방식을 활용하여 배열을 둘로 나누어 들어가면서 오름차순으로 merge하는 과정에서 inversion이 있는 것이 발견되면 그 숫자만큼을 inversion count로 더해주는 것이다. inversion counting은 merge과정에서만 일어나는데, 왼쪽과 오른쪽의 array는 모두 정렬된 상태라고 할 수 있다. 만약 왼쪽에 있는 array의 merge 차례가 된 index와 오른쪽에 있는 인덱스의 value가 뒤바뀌어 있다면 왼쪽 array의 나머지 value들도 모두 inversion이 되었다고 볼 수 있다.

이 때 각 divide하는 시간은 상수 시간, merge하는 시간은 배열의 처음부터 끝까지 루프를 돌면서 배열을 합치므로 divde와 merge하는 시간을 합쳐서 cn으로 책정하였다. T(n) = T(n/2) + T(n/2) + cn, T(1) = 1 으로 계산하면 O(nlogn)의 시간복잡도를 갖도록 할 수 있다.

**b) merge sort 변경**

우선 merge\_sort 함수에서 바뀐 것은 inversion counting 값을 리턴하기 위하여 리턴 타입을 void 에서 long long int로 바꾸었다. merge 함수 역시 long long int로 리턴값을 바꾸고, counted inversion을 리턴하여 준다.

merge에서는 inversion count를 구하기 위해 추가적인 코드가 삽입되었다. left array, right array에서 각각 index를 하나씩 증가시키며 merge하는 과정에서 위의 알고리즘 설계방식을 구현하기 위하여 left array index value < right array index value 이면 inversion += middle - left\_index + 1 로 알고리즘 코드를 완성하였다.