**Design and Analysis of Algorithms**

20161566 권형준

1. **실험 환경**

|  |  |
| --- | --- |
| CPU | Intel Core i5-8265U |
| CPU 속도 | 1.80 GHz |
| 메모리 용량 | 8.00 GB |
| 시스템 종류 | 64비트 운영체제, x64기반 프로세서 |
| OS 종류 | Windows 10 |

1. **4개의 알고리즘 비교 분석**

* Input이 Random일 경우(/s)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm  Input size(avg) | 1 | 2 | 3 | 4 |
| 10000 | 0.112116 | 0.002833 | 0.059124 | 0.003078 |
| 2^15=32768 | 0.923255 | 0.012247 | 0.164561 | 0.010895 |
| 2^18=262144 | 47.030744 | 0.071789 | 1.13166 | 0.06811 |
| 2^20=1048576 | 753.999175 | 0.289144 | 4.471555 | 0.271002 |

1. Input size가 10000(/s)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. Input size가 32768(/s)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. Input size가 262144(/s)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. Input size가 1048576(/s)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

* Input이 역으로 정렬 되어있을 경우(/s)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm  Input size | 1 | 2 | 3 | 4 |
| 10000 | 0.227487 | 0.003267 | 0.058518 | 0.002490 |
| 2^15=32768 | 1.746003 | 0.008781 | 0.179925 | 0.008161 |
| 2^18=262144 | 101.409535 | 0.081095 | 1.129096 | 0.052284 |
| 2^20=1048576 | 1626.942709 | 0.305965 | 4.785912 | 0.174166 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm  Input size(avg) | 1 | 2 | 3 | 4 |
| 10000 |  |  |  |  |
| 2^15=32768 |  |  |  |  |
| 2^18=262144 |  |  |  |  |
| 2^20=1048576 |  |  |  |  |

1. **알고리즘 4의 구조와 그렇게 설계한 이유**

수업시간에 introspective sort를 배웠다. Introspective sort는 처음에는 quick sorting을 실시하다가 만약 depth\_limit이 설정자가 설정한 값보다 더 낮아지게 되면 heap sorting을 하게 되고 마지막으로 정렬하는 부분 배열이의 길이가 50보다 작아지게 되면 insertion sorting을 하는 sorting algorithm이다. 이 아이디어와 merge sorting의 장점을 합쳐서 만들어진 함수가 best\_sorting algorithm이다. Introspective sort에서 heap sorting 대신 merge sorting을 사용하였다. Heap sorting은 항상 O(nlogn)의 시간 복잡도를 갖는 실용성이 높은 sorting 방법이다. 하지만 quick sorting보다 살짝 느리며, 데이터의 상태에 따라서 다른 O(nlogn)의 시간 복잡도를 가지는 sorting보다 조금 느린 편이다. 그리고 안전성을 보장받지 못하기도 한다. 이러한 단점을 보안하기 위해 merge sorting을 이용하면 어떨까 라는 생각을 하게 되었다. Merge sorting는 quick sorting과 비슷하게 원본 배열을 분할하며 정렬하는 방법이라 logN만큼의 시간이 걸리며, quick sorting은 pivot에 따라 시간 복잡도가 O(n^2)까지 나올 수 있는 반면 merge sorting은 heap sorting과 마찬가지로 O(nlogn)의 값으로 유지가 된다. Quick sorting과 같이 성능이 안 좋아지는 경우 또한 없다. 다만 추가적인 메모리가 필요하다는 점에서 효율성을 떨어뜨릴 수 있다. 이점을 보안하기 위해 n의 값을 최대한으로 quick sort를 사용하여 줄이고 나서 너무 많은 divide가 일어나는 경우, 즉 너무 많은 재귀함수가 호출되는 경우 merge sort를 이용한다. 최대한 quick sorting으로 진행을 하다가 만약 quick sorting으로 log(총 list의 길이)/log2의 깊이보다 더 깊게 들어갔다면 merge sorting을 사용하기 시작한다는 뜻이다. Log(총 list의 길이)/log2의 값은 처음부터 quick sort를 사용하지 않고 merge sorting을 사용하였을 때 호출되는 재귀함수의 수, 즉 들어가는 level의 수이다. Divide를 반복하다 보면 나오는 list의 길이가 50보다 작으면 selection sort를 실시하는데, 이는 O(n^2)이라는 시간 복잡도를 가진 selection sort이지만 실질 적으로는 데이터의 형태에 따라 그것보다 적은 시간 복잡도를 가지기 때문이며, 정렬된 list에서는 매우 빠른 시간 복잡도를 가지기 때문에 이를 사용한다. 이러한 특이 케이스에서 줄어드는 시간과, nlogn 과 n^2을 50이하에서 비교하면 특이 케이스의 경우에서 줄어드는 시간을 선택하는 게 더 이득이라는 결론을 내릴 수 있다. 또한 merge나 quick sort에 비해 추가적인 메모리의 요구나, 비교 operation의 수가 적기 때문에 작은 n에 관해서는 더 유리하다. 코드를 살펴보면 다음과 같다.

Void best\_sort(int A, int left, int right, int depth\_limit)

* A는 정렬하고자 하는 배열이고, left와 right은 정렬하고자 하는 구간이다. Depth\_limit은 quick sort에서 merge sort로 전환하는 경계선이다.

While (right-left > size\_threshold)

* 만약 구간의 길이가 size\_threshold(=50) 보다 작아질 경우 while문을 탈출하고 insertion sort를 실시한다. N읙 값이 작을 경우 insertion sort가 유리하기 때문이다.

If(depth\_limit==0) merge\_sort(A,left,right);

* 정해 놓은 depth\_limit이 0이 되었을 경우 quick sorting을 중지하고 merge sort를 실시한다. 초기 depth\_limit은 만약 처음부터 quick sorting 대신 merge sort를 사용했을 시 재귀 되는 함수의 수로, 그럴 경우의 depth라고 할 수 있다. Best\_sort를 재귀 호출할수록 depth\_limit은 1씩 작아진다.

Else

* Quick sort를 실시하는 부분이며 best\_sort를 재귀 호출한다. 재귀 호출은 한번으로 충분하다. 이 이유는 while문과 right에 temp, 즉 pivot, 을 입력해 줌으로서 재귀 호출의 효과를 내기 때문이다.

1. **실험 구조**

코드를 구현하는 데 사용된 함수는 총 8개이다. 우선 이 8개의 함수를 분석하면 다음과 같다.

* Void SWAP (int \*a, int \*b)

이 함수는 A와 B의 값을 받아서 둘의 값을 바꿔주는 함수이다. 이는 이 두 값의 본질적인 값을 바꿔야 함으로 포인터를 사용하여 함수내에서의 변수만 바꾸지 않고 함수를 호출한 변수의 주소값에 저장된 값을 바꿔 본질적인 값을 바꾼다. 이 함수는 make\_pivot에서 피벗과 A[right]의 값을 바꿔주는데 사용된다.

Int temp 🡪 변수를 임시 저장하기 위해 사용하였다.

* Int find\_pivot (int \*A, int left, int right)

이 함수는 A의 배열에서 가장 왼쪽 값과 오른쪽 값 그리고 중간 값을 비교하여 중간 값을 반환한다. 이 함수의 목적은 pivot으로 가장 알맞은 수를 찾아내는 것인데, 이 함수를 통해서 구한 pivot이 가장 최선의 pivot이라는 보장은 없으나 그럴 확률을 조금이라도 높이기 위해서 사용한다.

Middle 🡪 (right+left)/2 즉, 중간 값을 저장한다.

if문을 이용하여 세 값의 비교를 통해 크기의 중간 값을 반환한다.

* Int make\_pivot(int\* A, int left, int right)

이 함수는 find\_pivot을 통해서 정한 pivot을 사용하여 왼쪽에는 자신보다 작은 값을 오른쪽에는 자신보다 큰 값을 정렬한다. 우선 가장 오른쪽의 값과 pivot으로 정한 값을 SWAP해준다. 그 다음 가장 왼쪽부터 pivot의 값과 비교를 하면서 만약 작다면 SWAP한다. 마지막까지 비교를 완료하면 마지막으로 SWAP한 곳에 pivot의 값을 집어넣는다.

Pivot 🡪 pivot값보다 작은 값들을 어디까지 SWAP 해주기 위한 지표

For과 if문을 사용하여 A[i]가 A[pivot]보다 작으면, 즉 비교하는 값이 pivot값보다 작으면 SWAP을 하고, pivot++을 함으로서 만약 pivot값보다 작은 값이 또 있으면 다음 자리에 넣어 주게 된다. 이 과정을 마치면 A[pivot]의 좌측에는 pivot값보다 작은 값들 만 있게 된다.

* Void quick\_sort(int \*A, int left, int right)

Make\_pivot과 find\_pivot을 이용하여 quick sort를 실행하는 함수이다. 우선 right가 left보다 큰 경우에만 작동을 한다. Make\_pivot을 이용하여 우선 pivot을 기준으로 큰 값을 오른쪽 작은 값은 왼쪽으로 배치한다. 다음 각 배치한 좌측과 우측을 재귀함수를 이용하여 quick sorting을 실시한다.

* Void merge (int\* A, int left, int middle, int right)

merge함수는 merge sort에서 사용되며, 범위 내의 정렬된 두 배열을 정렬된 하나의 배열로 합치는 역할을 한다. 우선 temp라는 새로운 배열을 선언해서 정렬하고자 하는 배열의 범위인 left~right까지 temp에 넣어준다. 다음 i\_left와 i\_right 에 left 와 middle+1을 대입하는데, 이유는 현재 정렬된 배열은 left~middle 과 middle+1~right 까지 임으로 각각의 배열의 첫번째 index부터 비교를 하여 더 작은 값을 새로운 배열에 넣어주는 형식으로 하나의 정렬된 배열로 합치게 된다. 이를 while문을 통해 계속 temp[i\_left]와 temp[i\_right]을 비교해 주었으며, 만약 한 배열의 끝까지 비교가 완료되었다면, 그 뜻은 나머지 배열의 남은 값들은 현재 하나로 합쳐진 배열에 들어간 값들 보다 크다는 의미임으로 나머지를 순서대로 넣어주었다.

Int \*temp 🡪 sorting 하고자 하는 A를 임시 저장하는 배열

I\_left, i\_right 🡪 하나의 정렬된 배열로 합치기 이전의 두 배열의 가장 첫 index를 가리키게 초기 설정을 하고 하나씩 비교하기 위해 옮겨가는 값.

* Merge\_sort(int\* A, int left, int right)

merge함수를 이용하여 리스트를 정렬한다. Right가 left보다 클 동안 left~middle 그리고 middle~right를 merge sorting한다. 그리고 merge를 하여 가장 작은 단위까지 나누고 merge하면서 다시 리스트를 합치면서 정렬을 한다.

Int middle 🡪 (right + left)/2를 하여 중간 값을 저장한다.

* Void insertion\_srot(int \*A, int left, int right)

이 함수는 insertion sort를 하는 함수이다. 지정된 범위 내에서 index하나씩 살피면서 어느 자리에 들어갈지 비교를 하면서 정렬을 실행한다.

Int key 🡪 자리를 찾는 값, 가장 큰 수부터 비교를 통해 내려가다가 만약 아래의 값이 더 작다면 그 자리에 넣어준다.

* Void best\_srot(int \*A, int left, int right, int depth\_limit)

이 함수는 따로 설명을 하겠지만 자료구조만 보면 다음과 같다.

Int temp 🡪 quick sort를 실시할 때 pivot을 저장하는 값이다

While 🡪 left~right 까지의 범위가 size\_threshold(=50)보다 큰 경우에 반복한다. 만약 그 범위보다 작아지면 insertion sort를 진행한다.

If 🡪 depth\_limit 이 0이 아니면 quick sorting을 0이면 merge sorting을 실시한다.

* Main 함수

파일을 읽어서 algorithm index와 sorting하고자 하는 배열을 가지고 어떤 함수를 호출할지 결정한다. 결과를 output file을 열어 저장한다.

Int algorithm 🡪 algorithm index를 저장한다, 즉 insertion, quick, merge, best 중 sorting algorithm을 선택한다.

Int \*sort 🡪 정렬하고자 하는 배열을 저장한다.

Char filename[200] 🡪 input file의 이름을 저장한다.

Char result\_filename[207] 🡪 output file의 이름을 저장한다.

Clock\_t star\_time, end\_time 🡪 시작과 끝시간을 저장한다.

1. **결과 분석**

실험 결과 O(n^2)의 시간 복잡도를 가지는 insertion sort는 다른 sorting에 비해 눈에 띄게 느렸다. n의 값이 커질수록 차이가 더 심하게 나타나는 것을 알 수 있었다. 이는 다른 sorting의 시간 복잡도가 O(nlogn)이라는 점을 생각하면 logn 과 n의 값을 비교하는 것과 같으며, 둘의 기울기는 미분을 실시한 1/n과 1임으로 n의 값이 커지면 커질수록 logn은 기울기가 작아지지만 n은 꾸준히 1로 증가한다. 따라서 n^2과 nlogn의 차이도 n의 크기가 커지면 더 커짐을 알 수 있다. 하지만 Algorithm 2, 3, 4는 모두 O(nlogn)이라는 같은 시간 복잡도를 가지면서도 실행시간에 차이가 일어났다. 특히 merge sort는 quick과 직접 만든 best sorting에 비해서 2^20의 input이 들어갔을 시 10배 이상의 속도 차이가 났다. 이는 merge sort가 무조건 O(nlogn)이라는 시간 복잡도를 보장하면서도 그에 비해 너무 많은 공간 복잡도를 차지하기 때문이다. 매번 stack에서부터 데이트를 불러와야 하기 때문에 cache hit이 보장이 되지 않으며 데이터의 양이 많아지면 무시하지 못할 기능의 제약이 생기기 때문이다. 즉 merge sort는 데이터의 양이 방대해지면 quick sort보다 성능이 떨어질 가능성이 높아진다. 이런 점을 보안한 best sorting은 n의 값이 적당히 작아진 값으로 변하면 merge sorting을 실시하였고 그 결과 quick sorting 보다 약간씩 빠른 것을 볼 수 있다. 이는 n의 값이 커지면서 확실하게 확인할 수 있었으며 n의 값이 1000으로 아주 작을 경우에는 quick sort가 더 빠른 것을 확인하였다. Input이 역으로 정렬된 list인 경우 quick sort와 best sort의 시간 차이는 더 커졌는데, 이는 insertion sorting 덕분이다. Quick sorting에서 pivot을 처음과 마지막 그리고 중간 값을 비교하여 중간 값으로 설정하였기 때문에 best sort에서 merge sort는 호출되지 않았을 것이다. 이는 이미 반대로 정렬된 list가 들어오면서 merge와 qucik이 divide하는 기준이 같은 값이 되어 divide하는 횟수, 즉 depth가 똑같아 지면서 depth\_limit이 0이 되었을 시 이미 right-left가 size\_threshold(=50)보다 커서 insertion sorting이 실행되었을 것이다. 따라서 둘의 속도 차이는 insertion sorting에서 일어나는데, insertion sorting은 앞에서 설명한 바와 같이 n의 값이 작으면 operation수가 적고, 추가적인 공간 복잡도가 필요하지 않기 때문에 다른 O(nlogn)의 속도보다 빠르다. 이러한 이유 때문에 quick sort와 best sort에 역으로 정렬된 list가 입력되면 실행시간의 차이가 더 커짐을 유추할 수 있다.