Master of Sorting

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

void sort\_1(vector<double> &num, int l, int r)

* Insertion Sort 구현
* Input : num - 정렬 대상 1차원 배열, l - 시작 index, r - 끝 index
* Output : 없음
* Function :
  + 배열을 sorted 부분과 unsorted 부분으로 나누어 unsorted 부분의 가장 앞 값을 sorted 배열의 맞는 위치에 넣어준다.
  + 인자로 l, r 구간이 아닌 배열 항목의 개수를 넘겨줘도 되지만, sort\_4에서 특정 구간에 대해 insertion 을 하기 위해서 구간을 넘겨준다.
* 시간 복잡도 : O

void sort\_2(vector<double> &num, int l, int r)

* Quick Sort 구현
* Input : num - 정렬 대상 1차원 배열, l - 시작 index, r - 끝 index
* Output : 없음
* Function :
  + pivot을 선정하여 배열을 pivot보다 작은 부분, pivot보다 큰 부분으로 나누어 주고 각 부분에 대해 재귀적으로 호출을 한다.
  + 정렬이 된 배열에 대해서 최악 시간 복잡도를 갖는다.
* 시간 복잡도 : 평균 - O / 최악 – O

void sort\_3(vector<double> &num, vector<double> &temp, int l, int r)

* Merge Sort 구현
* Input : num - 정렬 대상 1차원 배열, temp – 정렬을 위해 값을 임시로 저장하는 1차원 배열

l - 시작 index, r - 끝 index

* Output : 없음
* Function :
  + 구간을 절반으로 나누고 재귀적으로 호출하여 각 구간을 먼저 정렬한 다음, 정렬된 두 구간을 동시에 앞에서부터 훑으며 더 작은 값을 선택하는 방식으로 정렬을 한다.
* 시간 복잡도 : O

void sort\_4(vector<double> &num, int l, int r)

* sort optimization
* Input : num - 정렬 대상 1차원 배열, temp – 정렬을 위해 값을 임시로 저장하는 1차원 배열

l - 시작 index, r - 끝 index

* Output : 없음
* Function :
  + Dual pivot, random pivot, size 별 최적화 등을 통해 시간을 줄인 정렬방법
* 시간 복잡도 : O

1. Random list - 함수 별 수행시간 비교

* Input data 설명
  + Size : 16, 256, 4096, 65536
  + Max value : 100000
  + Min value : -100000
  + Seed value : 141284, 141285, 141286, 141287, 141288
  + Data : 소수점 6자리 제한
  + 각 함수 별 4개의 사이즈에 대해 다른 5개의 Input Data로 시간을 측정한 뒤 평균을 내어 결과를 뽑아냈다.
  + 수행시간 비교 (단위 : s)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sort\_1 | Sort\_2 | Sort\_3 | Sort\_4 |
| 16 | 0.000005 | 0.000007 | 0.000006 | 0.000005 |
| 256 | 0.000354 | 0.000054 | 0.000078 | 0.000059 |
| 4096 | 0.076188 | 0.001398 | 0.001642 | 0.001290 |
| 65536 | 14.912304 | 0.029515 | 0.037585 | 0.029037 |

(사이즈의 변화에 시간의 변화가 매우 민감하므로 그래프는 제외하고 코멘트로 대체합니다.)

* Comments
  + 우선, 가장 작은 사이즈 n=16일 때는 시간의 차이가 거의 없고, 심지어 O(nlogn) 알고리즘인 2,3 번째 정렬이 시간이 더 걸리는 것을 확인할 수 있었다. 이는 재귀 호출에 의한 것으로 볼 수 있지만 그 정도도 매우 미미하므로 신경 쓰지 않아도 될 것 같다.
  + sort\_1의 경우, O(n^2)의 시간 복잡도를 갖는 Insertion sort를 이용했고, 각 사이즈가 16만큼 늘어날 때마다(16 -> 256 케이스 제외) 약 200배쯤 시간이 늘어나는 것을 확인할 수 있다. 배열의 구성과 상수배의 연산수가 같지 않으므로 차이가 같진 않다.
  + sort\_2, sort\_3의 경우, 각각 Quick sort, Merge sort를 이용하였고 random list Input이었으므로 Quicksort의 시간 복잡도를 O(nlogn)이라고 가정하고 두 함수 모두 256에서 4096으로, 4096에서 65536으로 데이터 사이즈가 16배만큼 증가했을 때 적게는 20배, 많게는 25배 이상 시간이 늘어나는 것을 확인하였다.
  + sort\_4의 경우에 최적화 전 정렬방법인 sort\_2와 비슷한 시간 복잡도를 갖는 것을 확인할 수 있다..

1. Non-Increasing list - 함수 별 수행시간 비교

* Input data 설명
  + Size : 16, 256, 4096, 65536
  + Max value : 100000
  + Min value : -100000
  + Seed value : 141284, 141285, 141286, 141287, 141288
  + Data : 소수점 6자리 제한
  + 각 함수 별 4개의 사이즈에 대해 다른 5개의 Input Data로 시간을 측정한 뒤 평균을 내어 결과를 뽑아냈다.
  + 수행시간 비교 (단위 : s)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sort\_1 | Sort\_2 | Sort\_3 | Sort\_4 |
| 16 | 0.000007 | 0.000006 | 0.000006 | 0.000007 |
| 256 | 0.000770 | 0.000418 | 0.000055 | 0.000024 |
| 4096 | 0.152309 | 0.117355 | 0.001652 | 0.000859 |
| 65536 | 28.760721 | 15.798647 | 0.020302 | 0.017406 |

(사이즈의 변화에 시간의 변화가 매우 민감하므로 그래프는 제외하고 코멘트로 대체합니다.)

* Comments
  + sort\_1에 구현한 Insertion sort의 경우에 sorted 부분과 unsorted 부분으로 나누어 unsorted 부분의 첫번째 값을 sorted array의 맞는 위치에 넣어준다. 그런데 정렬 전의 배열이 non-increasing 상태로 정렬되어 있으므로 unsorted 부분의 첫번째 값은 sorted 부분을 전부 탐색 후에 맞는 위치를 찾을 수 있다. 그러므로 연산은 최대인 에 거의 근접해진다. 즉, insertion sort의 최악 시간 복잡도를 갖는 케이스가 된다. 위에서 본 random list의 경우 때와 비교하면 약 2배의 시간이 나오는 것을 확인할 수 있다.
  + sort\_2에 구현한 quick sort의 경우에는 pivot을 선정 후, 구간을 두개로 나누어 재귀적으로 호출을 한다. 배열이 역으로 정렬되어 있는 경우에 O(n)으로 배열을 탐색할 때는 swap이 전혀 없고, 마지막에 pivot을 맞는 자리에 넣어줄 때, 한 번 swap이 일어난다. 그러고 나서 n-1 개의 항목에 대해 재귀호출이 일어난다. 즉, Insertion sort와 마찬가지로 에 연산이 가까워진다. 하지만 Insertion sort와 달리 swap 호출이 거의 없으므로 대략 절반정도의 수행시간이 나오는 것을 확인할 수 있다.
  + sort\_3의 경우는 random list 일 때보다 거의 절반의 시간이 걸린 것을 확인할 수 있는데 이를 분석해보면 구간을 두개로 나누고 재귀호출이 끝난 뒤, 두 구간을 합치는 과정에서 우측 값들이 모두 왼쪽 값들보다 작기 때문에 우측 값을 먼저 배열에 전부 넣고, 그 이후 좌측 값을 전부 넣는다. 즉, 어셈블리 상에서 index 값의 변화를 최소화로 할 수 있기 때문이라고 결론 내릴 수 있었다.
  + sort\_4는 시간 복잡도가 O 으로 안정되었으므로 같은 quick sort인 sort\_2와 달리 sort\_3보다 더 적은 시간이 걸리는 것을 확인할 수 있었다. 또, 과 의 차이로 sort\_3보다 수행 시간이 덜 걸리는 것을 결론 내릴 수 있었다.

1. Non-Decreasing list - 함수 별 수행시간 비교

* Input data 설명
  + Size : 16, 256, 4096, 65536
  + Max value : 100000
  + Min value : -100000
  + Seed value : 141284
  + Data : 소수점 6자리 제한
  + 수행시간 비교 (단위 : s)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sort\_1 | Sort\_2 | Sort\_3 | Sort\_4 |
| 16 | 0.000003 | 0.000008 | 0.000007 | 0.000004 |
| 256 | 0.000004 | 0.000890 | 0.000073 | 0.000033 |
| 4096 | 0.000067 | 0.149690 | 0.001230 | 0.001096 |
| 65536 | 0.000653 | 25.555665 | 0.027411 | 0.018702 |

* Comments
  + sort\_1에 구현된 Insertion sort의 최악 시간 복잡도인 경우가 역으로 정렬된 상태였다면 반면에 정방향으로 정렬된 배열은 가장 적은 시간인 O(n)의 시간 복잡도에 정렬을 마무리 할 수 있다. 그렇기 때문에 65536 사이즈의 Input data에 대해서도 1ms가 걸리지 않는 것을 확인할 수 있다.
  + sort\_2에서는 위의 non-increasing input 과 마찬가지로 O의 시간 복잡도를 갖는데 O(n)의 시간으로 배열을 탐색할 때, 모든 값을 swap 하기 때문에 그만큼 연산 수가 늘어나 sort\_2보다는 sort\_1에 더 근접한 결과를 갖는 것을 확인할 수 있다.
  + sort\_3와 sort\_4 는 non-increasing 과 큰 차이가 없는 것을 확인하였다.

1. sort\_4 최적화 방법

* 최적화 1) Random pivot
  + 위의 케이스 분류에서 설명했듯이 Quick sort 는 역정렬, 정렬 모두 O의 시간 복잡도를 갖는 것을 확인할 수 있었다. 그렇기 때문에 특이 케이스에서 pivot 선택이 문제가 되는 경우를 해결하기 위해 srand, rand 함수를 이용해 매번 pivot을 임의로 선택하게 하였다. 이로써 구간을 나누는 과정에서 구간의 치우침을 어느정도 해결하였다.
* 최적화 2) Dual pivot
  + 기존 Quick sort에서 single pivot으로 구간을 2개로 나누는데 sort\_4에서는 dual pivot을 이용하여 구간을 3개로 나누었다. single pivot을 통해 구간이 항상 절반으로 나뉘었다고 했을 때, 시간 복잡도는 O로 결정된다면 3개로 나뉜다면 O으로 결정된다. 큰 차이는 없지만 상수배 만큼의 시간을 절약할 수 있는 효과가 있다.
* 최적화 3) 작은 사이즈 문제 해결
  + 위에서 다룬 대부분의 케이스 중 16 사이즈의 sort\_1과 sort\_2의 수행시간을 보면 sort\_1의 수행 시간이 더 적은 것을 볼 수 있다. 재귀 호출 등으로 인한 시간 증가로 볼 수 있는데 그리하여 작은 사이즈에서는 insertion sort를 호출하여 해결하도록 하였다.