|  |
| --- |
|  |
| 알고리즘설계와분석 <HW3> |
|  |
|  |
| 컴퓨터공학과 20161620 이수연 |
| **CSE3081-02 임인성 교수님** |
| **2017. 11. 26. (일)** |

**1. 과제 수행 환경**

* OS : Windows 8.1 Enterprise K
* RAM : 4.00GB
* CPU : Intel® Pentium® 3805U @ 1.90GHz 1.90GHz
* 시스템 종류 : 32비트 운영 체제, x64 기반 프로세서
* Compiler : visual studio 2017 Release Mode

**2. 각 구현 함수의 실행 시간 측정**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **(시간단위 = ms)** | **Insertion sort** | **Heap sort** | **Qsort()** | **Quick\_sort** | **Quick\_sort\_opt** |
| **N = 32 (2^5)** | 0.018 | 0.034 | 0.103 | 0.248 | 0.021 |
|  | 0.018 | 0.040 | 0.063 | 0.223 | 0.017 |
|  | 0.015 | 0.052 | 0.058 | 0.217 | 0.018 |
|  | 0.022 | 0.037 | 0.058 | 0.221 | 0.045 |
|  | 0.019 | 0.042 | 0.057 | 0.227 | 0.022 |
| **평균** | **0.0184** | **0.0410** | **0.0678** | **0.2272** | **0.0246** |
| **N = 1024 (2^10)** | 8.847 | 0.935 | 2.267 | 13.595 | 6.357 |
|  | 7.035 | 1.032 | 2.751 | 10.294 | 5.962 |
|  | 8.254 | 1.020 | 2.727 | 11.048 | 5.376 |
|  | 7.772 | 0.892 | 2.680 | 10.480 | 6.002 |
|  | 8.028 | 0.992 | 3.327 | 11.368 | 5.872 |
| **평균** | **7.9872** | **0.9742** | **2.7504** | **11.357** | **5.9138** |
| **N = 16384 (2^14)** | 1652.548 | 72893.342 | 66.914 | 243.695 | 2120.339 |
|  | 1768.823 | 84325.489 | 69.425 | 230.449 | 2204.314 |
|  | 1650.099 | 83411.235 | 72.953 | 221.495 | 2001.486 |
|  | 1737.404 | 70901.284 | 69.508 | 218.110 | 1996.223 |
|  | 1742.762 | 82398.783 | 66.657 | 240.293 | 2185.996 |
| **평균** | **1710.3272** | **78786.0266** | **69.0914** | **230.8084** | **2101.6716** |
| **N = 131072 (2^17)** | 87743.234 | **측정불가** | 655.955 | 1865.682 | 70145.320 |
|  | 79445.422 | 596.386 | 1918.079 | 59923.008 |
|  | 88345.545 | 635.336 | 2281.140 | 62387.432 |
|  | 80237.405 | 796.315 | 2055.771 | 59434.993 |
|  | 87209.122 | 794.225 | 1979.761 | 68723.148 |
| **평균** | **84596.1456** | **695.6434** | **2020.0866** | **64122.7808** |
| **N = 1048576 (2^20)** | **측정불가** | **측정불가** | 5475.568 | 19305.945 | **측정불가** |
|  | 6039.794 | 17874.441 |
|  | 5365.555 | 11979.478 |
|  | 5619.596 | 15630.221 |
|  | 5255.064 | 11586.200 |
| **평균** | **5551.1154** | **15275.257** |

**3. 실제 실행 시간과 이론적인 시간복잡도간의 연관성**

**1. insertion sort**

- 이론적인 시간복잡도 : O(n^2)

- 실제 실행 시간 :

|  |  |
| --- | --- |
| **N = 32 (2^5)** | **0.0410** |
| **N = 1024 (2^10)** | **7.9872** |
| **N = 16384 (2^14)** | **1710.3272** |
| **N = 131072 (2^17)** | **84596.1456** |
| **N = 1048576 (2^20)** | **측정불가** |

표에서 보이는 것과 같이 N이 커질수록 N^2에 맞추어 (정비례 X) 기하급수적으로 실행 시간이 증가함을 알 수 있다. N이 2^20일 때는 결과가 나오는데 너무 오랜 시간이 걸려 ‘측정불가’ 로 기록하였다.

**2. heap sort**

- 이론적인 시간복잡도 : O(nlogn)

- 실제 실행 시간 :

|  |  |
| --- | --- |
| **N = 32 (2^5)** | **0.0184** |
| **N = 1024 (2^10)** | **0.9742** |
| **N = 16384 (2^14)** | **78786.0266** |
| **N = 131072 (2^17)** | **측정불가** |
| **N = 1048576 (2^20)** | **측정불가** |

Heap sort의 이론적인 시간복잡도는 O(nlogn)이다. 이론적으로 볼 때, logn은 보통 n보다 작기 때문에 시간복잡도가 O(n^2)인 insertion sort보다는 시간이 적게 걸려야 했다. 그러나 위의 표와 같이 N의 값이 작을 때는 (32, 1024일 때) 확연하게 insertion sort보다 시간이 적게 측정되었으나, N이 커지자 갑자기 측정 시간이 증가하여 결국 두 항목이나 시간 측정이 불가하였다. 이는 이론적인 시간복잡도와는 연관성이 떨어지는 결과였다.

**3. qsort() – c library 함수 이용**

- 이론적인 (평균) 시간복잡도 : O(nlogn)

- 실제 실행 시간 :

|  |  |
| --- | --- |
| **N = 32 (2^5)** | **0.0678** |
| **N = 1024 (2^10)** | **2.7504** |
| **N = 16384 (2^14)** | **69.0914** |
| **N = 131072 (2^17)** | **695.6434** |
| **N = 1048576 (2^20)** | **5551.1154** |

표에서 보이는 것과 같이 N이 커질수록 NlogN에 맞추어 (정비례 X) 실행 시간이 증가하였다. 보통 logn은 n보다 작으므로 이론적으로 insertion sort보다 증가 폭이 점진적으로 작아야 했는데 그 점도 일치하였다.

**4. quick\_sort**

- 이론적인 (평균) 시간복잡도 : O(nlogn)

- 실제 실행 시간 :

|  |  |
| --- | --- |
| **N = 32 (2^5)** | **0.2272** |
| **N = 1024 (2^10)** | **11.357** |
| **N = 16384 (2^14)** | **230.8084** |
| **N = 131072 (2^17)** | **2020.0866** |
| **N = 1048576 (2^20)** | **15275.257** |

실제 시간 측정 결과는 c 라이브러리 함수인 qsort()로 만들었을 때와 같이 N이 커질수록 점진적으로 증가하였다. 하나씩 비교하여 적당한 pivot을 뽑은 후 그 과정에서 정렬하는 방법을 이용하였고, qsort()보다는 조금 더 오래 걸렸다.

**5. quick\_sort\_opt**

- 이론적인 (평균) 시간복잡도 : O(nlogn)

- 실제 실행 시간 :

|  |  |
| --- | --- |
| **N = 32 (2^5)** | **0.0246** |
| **N = 1024 (2^10)** | **5.9138** |
| **N = 16384 (2^14)** | **2101.6716** |
| **N = 131072 (2^17)** | **64122.7808** |
| **N = 1048576 (2^20)** | **측정불가** |

이 함수는 quick\_sort를 최적화 한 함수로, 대부분의 원소를 quick sort로 정렬한 후 insertion sort로 나머지를 정렬하는 방법을 사용하였다. 이론상으로 N의 크기가 작을 때는 다른 정렬 방법보다 insertion sort가 효율적인 것으로 알려져 있다. 따라서 이렇게 하면 모든 원소를 quick sort로 짰을 때 보다 이론적 시간이 적어야 한다. 표를 보면 실제로도 N이 32일 때와 1024일 때 quick sort보다 2-5배 가까이 빨라졌음을 알 수 있다. 그러나 N이 커지자 측정 시간이 급수적으로 커져 원래의 quick sort보다 훨씬 오래 걸렸고, 마지막 항목은 아예 측정이 불가했다. Insertion sort는 시간복잡도가 O(n^2)이기 때문에, 원소의 개수가 많아지자 오히려 이 방법을 썼을 때 비효율적이 되었다.

위의 항목들을 종합해보면 heap sort를 제외하고는 이론적인 시간복잡도와 실제 측정 시간간에 연관이 있고, 비슷한 경향을 나타낸다는 것을 알 수 있다.

**4. 서로 다른 종류의 데이터에 대한 수행 시간 차이**

서로 다른 종류의 데이터일 때 수행 시간에는 차이가 있다. 여기서 서로 다른 종류의 데이터란 원소의 개수 N이 다른 데이터 집합이 아니라, 원소의 수가 같아도 최초 배열이 다른 데이터 집합을 말한다. 가령 entirely random한 데이터 집합일 때와 descending한 데이터 집합 (정반대로 정렬)일 때 측정 시간은 상당히 달라진다. Insertion sort나 heap sort의 경우 정렬하려는 목적과 반대인 배열이 들어올수록 데이터를 서로 비교하는 횟수 (compare time)와 메모리를 할당하는 횟수 (assignment time)가 증가하기 때문이다. quick sort의 경우에는 최초 배열에서 pivot을 어떻게 뽑느냐에 따라 이론적인 시간복잡도가 평균적으로 O(nlogn), 최악의 경우 O(n^2)으로 구분된다. 시간복잡도에서 N이 커질수록 O(nlogn) 과 O(n^2)은 큰 차이가 있기 때문에 서로 다른 종류의 데이터에 대해서 수행 시간이 달라짐은 분명하다.

**5. quick sort는 정말 빠른지의 여부**

2번과 3번 항목을 보면 알 수 있듯이 quick sort는 다른 정렬 방법에 비해 실제로 빠르다. 이는 N이 커질수록 분명하게 드러난다. 특히 N의 크기가 2^20일 때는 insertion sort와 heap sort에서 모두 측정이 불가한 반면 quick sort에서는 나머지 두 정렬이 더 작은 N에 대해 실행한 시간보다도 훨씬 빨리 실행되었다. 이러한 점을 볼 때 quick sort는 이름대로 정말 빠르고 효율적인 정렬 방법이라고 할 수 있다.

**6. 어떤 부류의 데이터에 대해 insertion sort가 효율적인가**

Insertion sort는 전체 원소를 훑지 않고 배열의 일부분만 확인해도 정렬이 가능하다. 따라서 N의 크기가 작은 데이터일수록, 또한 특히 배열의 일부분이 정렬되어 있는 데이터에 대하여 그 효율성이 발휘된다. 2번과 3번 항목에서 실제로 N이 32일 때는 거의 선형적인 시간에 실행된 것을 알 수 있다.

**7. 어느 정도 범위의 n값까지 quick sort보다 빠르게 insertion sort를 이용할 수 있는가**

2번과 3번 항목에서 보면 N이 32 (2^5)일 때는 insertion sort가 빨랐으나 N이 1024(2^10)일 때는 quick sort가 확연히 빨랐다. 이 점에서 착안하여 N=64, 128, 256, 512일 때 실험을 해 본 결과 평균을 낸 시간으로 볼 때는 N=128 (2^7)일 때 까지는 insertion sort가 더 빨랐다. 따라서 N이 128이하 일 때는 insertion sort가 더 효율적이라고 할 수 있다. 하지만 각각의 실행에서 봤을 때 어떤 경우에는 N이 128일 때 quick sort가 더 빠르거나, 혹은 N이 512일 때 insertion sort가 더 빠르거나 하는 등 결과가 정확히 일정하지는 않았다.

**8. quick sort에서 재귀적으로 생성되는 트리의 깊이**

Entirely random 데이터에 대하여 실험을 해 본 결과 quick sort에서 재귀적으로 생성되는 트리의 깊이(depth)는 다음과 같았다.

- 평균적으로 시간복잡도가 O(nlogn)인 데이터 : depth는 O(logn)

- 시간복잡도가 O(n^2)인 데이터 (worst) : depth는 O(n)

이번에 수행한 과제를 통해 실험한 바로는 depth는 이론적으로 평균값인 O(logn)에 가까웠다. 하지만 pivot을 잘못 뽑는 경우, 특히 그저 random으로 pivot을 뽑거나 데이터 배열의 처음 값, 중간 값, 끝 값 중에서 pivot을 뽑는 quick sort의 경우엔 pivot이 치우쳐 worst 경우가 될 수 있다. 그런 경우에는 재귀적으로 생성되는 트리의 깊이도 O(n)이 될 것이다.

**9. 각 방법을 최대한 효율적으로 구현하기 위하여 어떤 노력을 기울였는가**

우선 데이터 이동이나 복사에 있어서 반복문을 사용하지 않고 memcpy함수를 이용하였다. Quick sort를 구현함에 있어서 pivot을 단순히 랜덤으로 뽑지 않고 하나씩 비교해가며 중간 값을 찾도록 했다. Quick sort를 최적화 할 때는 quick sort를 우선 이용하여 대부분 정렬해 놓은 뒤 insertion으로 나머지를 배열하는 방법을 이용하였다.

**10. 이번 숙제를 하면서 적용한 창의적인 방법이나 경험적으로 알게 된 중요사항을 서술하라**

저번 숙제를 할 때는 시간을 측정하는 방법을 제대로 터득하지 못해서 굉장히 어려운 함수를 참고해서 썼었는데 이번에 제시된 파일에서 예시를 보면서 제대로 된 시간측정 방법을 알게 되었다. 또한 데이터를 이동하거나 특히 복사할 때 초반엔 반복문을 이용했었는데, string에 포함된 memcpy 함수를 이용하면 편하고 효율적으로 할 수 있다는 것을 알게 되었다. c언어 라이브러리 자체에 sort를 진행하는 함수가 있다는 것도 알고 실제로 이용도 해 볼 기회가 있어 좋았다. 라이브러리 자체로 실행한 quick sort가 비교적 빠르게 실행되어서, 라이브러리에 적힌 코드와 나의 코드를 비교해보며 어떻게 더 효율적으로 quick sort를 설계할지 생각해 볼 수 있었다.

실행 결과를 통해 이론적으로 배운 시간복잡도를 확인할 수 있어 좋았다. 이번 실험에서도 여전히 N이 커지면 측정시간이 너무 길어져 도중에 그만두어야 할 때가 있었는데, quick sort를 이용하니 현저히 빨라지는 것을 직접 보면서, 시간복잡도를 조금이라도 작게 설정하는 것이 중요하다는 것을 다시 한번 되새길 수 있었다.

**(끝)**