CSE3081 (2반): 알고리즘 설계와 분석 HW2 보고서

20190258 김혜린

# 실험 환경

* **Operation System:** Windows 10 Pro 64-bit
* **Compiler:** Microsoft Visual Studio Community 2019
* **CPU:** Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz
* **RAM:** 32.0GB(31.9GB 사용 가능)

# 샘플 데이터

* 데이터 개수: 4096

## 데이터 생성 알고리즘

데이터 개수 n에 대해 0부터 n-1까지 iteration을 통하여 데이터를 생성한 후, Fisher-Yates shuffle algorithm을 이용해 데이터를 랜덤하게 섞어 예시 데이터를 완성한다.

* 배열 원소 타입: ELEMENT
* ELEMENT 멤버

i번째 iteration에 대해

* 1. **score :** int 타입, i를 저장한다.
  2. **data[3] :** float 배열 타입, 순서대로 i/n, -i/n, 2.0\*i/n을 저장한다.
  3. **comments[16] :** char 배열 타입, i를 4진수로 저장한다.

# 실험 결과

각 차시의 실험에 사용한 input data는 동일하며 각 함수의 prototype은 void fucntion(void\*, size\_t, size\_t, \_Cmpfun\*); 으로 동일하다.

함수의 parameter type 중 \_Cmpfun\*의 경우 int \_Cmpfun(const void\*, const void\*); 의 prototype을 가지며 이번 숙제에서는 ELEMENT 멤버 중 score을 기준으로 정렬을 진행하므로 int score\_comparison (const void\*, const void\*); 의 prototype을 가진 함수를 정의해 parameter로 넘어오는 원소 포인터 a와 b에 대해 score 값을 비교하여 a가 더 클 경우 1, 같을 경우 0, 작을 경우 -1을 반환하도록 함수를 구현하였다.

## 1번 함수 qsort()

1. 수행시간

1차: ELEMENT type of size 32: Time taken by qsort() = 1.723ms

2차: ELEMENT type of size 32: Time taken by qsort() = 1.718ms

3차: ELEMENT type of size 32: Time taken by qsort() = 1.893ms

1. 함수 알고리즘

C/C++ library에서 제공하는 qsort함수를 사용하였다.

## 21번 함수 qsort\_orig()

1. 수행시간

1차: ELEMENT type of size 32: Time taken by qsort\_orig() = 5.108ms

2차: ELEMENT type of size 32: Time taken by qsort\_orig() = 5.723ms

3차: ELEMENT type of size 32: Time taken by qsort\_orig() = 5.039ms

1. 함수 알고리즘
   1. 인자로 넘어오는 배열의 가장 왼쪽 함수를 pivot element로 하여 l\_ptr과 r\_ptr을 선언해 배열의 양 끝에서부터 각각 size만큼 더하거나 빼며 pivot보다 작거나 큰 동안 while 문을 진행하였다.
   2. l\_ptr의 경우, pivot보다 클 경우(\_Cmpfun 결과 1 반환), r\_ptr의 경우, pivot보다 작을 경우(\_Cmpfun 결과 1 반환) while문을 탈출하고 l\_ptr과 r\_ptr이 가리키는 원소를 swap\_ele(l\_ptr, r\_ptr); 을 통해 바꿔주어 pivot의 왼쪽에는 pivot보다 작은 원소들이, pivot의 오른쪽에는 pivot보다 큰 원소들이 올 수 있도록 해주었다.
   3. swap\_ele(l\_ptr, r\_ptr); 의 경우, ELEMENT 타입의 원소를 가리키는 임의의 변수 tmp\_ele을 동적할당해, 해당 변수에 l\_ptr을 복사한 뒤, l\_ptr에 r\_ptr을 다시 r\_ptr에 tmp\_ele을 복사하는 방식으로 진행된다.
   4. pivot을 이용한 정렬이 끝나는 조건은 l\_ptr과 r\_ptr이 모두 pviot을 가리키는 즉, l\_ptr과 r\_ptr이 일치할 때인데 pivot의 위치와 정렬 전 데이터의 상태에 따라 l\_ptr 또는 r\_ptr 중 한 변수만 pivot과 일치하는 경우가 발생할 수 있다. 이런 경우에서 아직 일치하지 않은 변수가 pivot을 기준으로 좌우 위치를 옮겨가야 하는 경우가 발생할 경우, l\_ptr과 r\_ptr을 이용해 swap\_ele 함수를 진행하고, 이 경우 pivot의 위치가 바뀌는 것이기 때문에 이를 반영해주었다.
   5. 현재 pivot과 인자로 넘어온 base의 차이를 size로 나누어 왼쪽 배열의 크기를 구하고(num\_l), pivot 1개와 num\_l을 제외한 나머진 개수를 num\_r로 하여 재귀적으로 다시 함수를 호출하였다.
   6. 재귀가 끝날 조건은 인자로 넘어오는 num(부분 배열의 원소 개수) 가 1보다 작거나 같을 때이다.
2. 비용 분석
   1. C/C++ library에서 제공하는 qsort함수와의 계산 비용을 분석해보면 직접 구현한 qsort\_orig함수가 월등히 느린 것을 확인할 수 있다.
   2. qsort\_orig 함수의 Order 시간 복잡도를 분석해보면 pivot에 대해 정렬할 때 배열 크기 num만큼의 O(num)이 소요되고, pivot에 대해 분할된 배열에 대해 재귀적으로 정렬이 진행되므로 input data n에 대해 평균적으로 O(nlogn)의 시간복잡도가 소요됨을 알 수 있다.
   3. O(nlogn)의 시간복잡도는 quick sort의 일반적인 시간복잡도와 동일하다.
   4. qsort함수와 비교했을 때 수행 속도가 느린 이유는 input n에 대한 처리 틀보단 그 안의 세부적인 구현으로 인해 수행 속도가 느려졌다고 결론 내릴 수 있다.
   5. 구현된 큰 틀 안에서 단순히 포인터에 대해 size만큼 왔다 갔다하는 것이 아닌 좀 더 효율적으로 pivot에 대한 정렬을 선형적으로 구현할 수 있는 방법이 필요한 것으로 분석된다. 또한 중간에 pivot과 l\_ptr과 r\_ptr이 일치할 때의 처리 방식에 대해 pivot의 위치를 바꾸는 것이 아닌 좀 더 효율적인 방법을 구상해낼 필요가 있다고 생각된다.

## 22번 함수 qsort\_median\_insert()

1. 수행시간

1차: ELEMENT type of size 32: Time taken by qsort\_median\_insert() = 3.972ms

2차: ELEMENT type of size 32: Time taken by qsort\_median\_insert() = 4.085ms

3차: ELEMENT type of size 32: Time taken by qsort\_median\_insert() = 4.035ms

1. 함수 알고리즘
   1. qsort\_orig 함수와 비교했을 때 두 가지가 바뀌어 있다. pivot 선택 strategy와 MIN\_NUM보다 작은 부분 배열에 대한 처리 방식이다.
   2. pivot strategy의 경우 배열의 가장 왼쪽, 오른쪽, 중앙 원소를 ordering한 후에 중앙 원소를 pivot으로 하여 qsort\_orig와 동일한 방식으로 pivot에 대한 정렬을 진행한 후, 부분 배열에 대해 재귀적으로 함수를 호출한다.
   3. 다만, 함수가 호출되고 인자로 넘어온 num이 MIN\_NUM보다 작다고 판단될 경우, quick sort를 진행하지 않고 insertion sort를 진행한다.
2. 비용 분석
   1. qsort\_orig함수보다는 빠른 계산 비용을 보이지만 qsort와 비교했을 때 여전히 느린 것을 확인할 수 있다.
   2. 이는 qsort\_orig에서 분석한 이유와 같은 이유로 분석된다.
   3. qsort\_orig 함수와 비교했을 때 비용이 꽤 감소한 것으로 보아 세 원소 중 median 값으로 pivot을 설정할 경우, 확률 상 부분 배열이 치우치지 않을 확률이 높아지고 pivot에 대한 정렬로 어느 정도 정렬된 작은 크기의 원소를 가진 배열에 대해 선형 시간의 시간 비용이 소요되는 insertion sort를 이용함으로써 계산 비용이 줄어들었다고 분석할 수 있다.

## 23번 함수 qsort\_median\_insert\_iter()

1. 수행시간

1차: ELEMENT type of size 32: Time taken by qsort\_median\_inset\_iter() = 3.909ms

2차: ELEMENT type of size 32: Time taken by qsort\_median\_inset\_iter() = 3.844ms

3차: ELEMENT type of size 32: Time taken by qsort\_median\_inset\_iter() = 3.957ms

1. 함수 알고리즘
   1. qsort\_median\_insert() 함수와 동일한 pivot 선택 strategy를 이용하며 위의 두 함수와 pivot에 대해 정렬하는 방식은 동일하다.
   2. 다만 pivot에 대해 정렬이 완료되고 분할된 두 배열에 대해 배열 크기가 더 작은 부분 배열만 재귀적으로 함수를 진행하고 다른 부분 배열의 경우 반복을 통해 quick sort를 진행한다.
   3. 반복을 통한 quick sort 진행을 위해 start 라는 포인터를 이용해 매 iteration에 대한 배열의 시작점을 저장하고 iteration이 끝났을 때 새로운 iteration에 대한 l\_ptr, start, pivot, r\_ptr을 재설정하는데 사용하였다.
   4. 더 이상 재귀적으로 함수를 호출하지 않는 기준은 qsort\_median\_insert 함수와 동일하게 인자로 넘어온 배열의 크기가 MIN\_NUM보다 작은 경우이며 이때 insertion sort를 이용해 작은 크기의 배열에 대한 정렬을 진행한다.
   5. iteration의 경우도 마찬가지로 재설정된 배열의 크기가 MIN\_NUM보다 작을 경우 iteration을 종료하며 해당 iteration의 시작점으로 설정된 start와 배열 크기 n에 대해 insertion sort를 진행한다.
2. 비용 분석
   1. qsort보다 느리고 qsort\_orig, qsort\_median\_insert보다는 빠른 것을 확인할 수 있다. qsort보다 느린 이유는 앞의 두 함수와 동일한 이유를 가지는 것으로 분석할 수 있다.
   2. qsort\_orig보다는 월등히 빠른 것을 확인할 수 있고 qsort\_median\_insert보다 약간 빠른 것을 확인할 수 있는데 이는 qsort\_median\_insert에서 기술한 이유에 더해 충분이 큰 입력 데이터 n에 대해 재귀 호출의 횟수를 줄이고 작은 크기의 부분 배열에 대한 재귀를 진행함으로써 stack의 사용을 감소시켰기 때문으로 분석할 수 있다.