CSE3081 알고리즘설계와분석 HW2 보고서

컴퓨터공학과 2 학년 20181662 이건영

1. 실험 환경

Operation System : Microsoft Windows 10 Home (64-bit)

Compiler : Microsoft Visual Studio Community 2019

CPU : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.80 GHz

RAM : 8.00GB

2. 코드 설명

1) cmp 함수

```
int cmp(const void* a, const void* b)
{
    ELEMENT* ele_a, * ele_b;

    ele_a = (ELEMENT*)a;
    ele_b = (ELEMENT*)b;

    if (ele_a->score == ele_b->score)
        return 0;
    else if (ele_a->score < ele_b->score)
        return -1;
    else return 1;
}
```

전달받은 두 인자의 대소를 비교하여 같으면 0, a 가 b 보다 작으면 -1, 크면 1 을 반환한다.

2) init_array 함수

```
void init_array(ELEMENT* array, int n)
{
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (int o = 0; o < 10; o++)
        {
            array[i].comments[o] = 1;
        }
        array[i].score = i;
        for (int k = 0; k < 3; k++)
        {
            array[i].data[k] = (float)k;
        }
    }
    //Shuffle by Fisher-Yates shuffle algorithm.
    srand((unsigned int)time(NULL));

    for (i = n - 1; i >= 1; i--)
    {
        j = rand() % (i + 1);
        {
            char buf[256];
            char* r_i = (char*)(array + i);
            char* r_j = (char*)(array + j);

            size_t m, ms;

            for (ms = sizeof(ELEMENT); ms > 0; ms -= m, r_i += m, r_j += m)
            {
                m = ms < sizeof(buf) ? ms : sizeof(buf);
                memcpy(buf, r_i, m);
                memcpy(r_i, r_j, m);
                memcpy(r_j, buf, m);
            }
        }
    }
}
```

ELEMENT 타입으로 이루어진 배열을 초기화하고 Fisher-Yates 알고리즘을 사용하여 값을 임의로 설정한다.

3) qsort_orig 함수

```
void qsort_orig(void* array, size_t n, size_t size, _Cmpfun* cmp)
{
    while (1 < n)
    {
        int i = 0;
        int j = n - 1;
        char* qs = (char*)array;
        char* qi = (char*)array;
        char* qj = qi + size * j;

        {
            char    buf[MAX_BUF];
            char* q1 = qi;
            char* q2 = qj;
            size_t  m, ms;

            for (ms = size; 0 < ms; ms -= m, q1 += m, q2 += m) {
                m = (ms < sizeof(buf)) ? ms : sizeof(buf);
                memcpy(buf, q1, m);
                memcpy(q1, q2, m);
                memcpy(q2, buf, m);
            }
        }
        //changed first and last value for pivot setting
        char* qp = qj;

        while (i < j)
        {

            while (i < j && (*cmp)(qi, qp) <= 0)
                ++i, qi += size;
            while (i < j && (*cmp)(qp, qj) <= 0)
                --j, qj -= size;

            if (i < j) {
                char    buf[MAX_BUF];
                char* q1 = qi;
                char* q2 = qj;
```

기본적으로 제공받은 qsort_example 함수와 비슷하게 진행되나, pivot 설정 시 배열의 가장 왼쪽 원소를 사용하기 위해 가장 왼쪽에 있던 원소를 가장 오른쪽의 값과 바꿔 주는 과정을 거쳤다. 이를 통해 가장 왼쪽의 원소를 pivot 으로 사용한 것과 같은 결과를 낼 수 있다.

```
            char* q1 = qi;
            char* q2 = qj;
            size_t  m, ms;

            for (ms = size; 0 < ms; ms -= m, q1 += m, q2 += m) {
                m = (ms < sizeof(buf)) ? ms : sizeof(buf);
                memcpy(buf, q1, m);
                memcpy(q1, q2, m);
                memcpy(q2, buf, m);
            }
            ++i, qi += size;

        }
    }

    if (qi != qp) {
        char    buf[MAX_BUF];
        char* q1 = qi;
        char* q2 = qp;
        size_t  m, ms;

        for (ms = size; 0 < ms; ms -= m, q1 += m, q2 += m) {
            m = (ms < sizeof(buf)) ? ms : sizeof(buf);
            memcpy(buf, q1, m);
            memcpy(q1, q2, m);
            memcpy(q2, buf, m);
        }
    }


    j = n - i - 1, qi += size;
    if (j < i) {
        if (1 < j)
            qsort(qi, j, size, cmp);
        n = i;
    }
    else {
```

이 후의 과정 역시 qsort_example 과 비슷하게 진행된다.

```
    else {
        if (1 < i)
            qsort(array, i, size, cmp);
        array = qi;
        n = j;
    }
```

4) median 함수

```
25  int median(size_t left, size_t right)
26  {
27      size_t a, b, c;
28      a = rand() % (right - left) + left;
29      b = rand() % (right - left) + left;
30      c = rand() % (right - left) + left;
31
32      if (a >= b && a >= c) return a;
33      else if (b >= a &&b >= c) return b;
34      else return c;
35  }
```

pivot 설정을 위한 median 함수이다. 전달받은 값 사이에서 임의의 세 정수를 생성해 그 중앙값을 반환한다.

5) insertion_sort 함수

```
10  void insertion_sort(void* array, size_t n, size_t size, _Cmpfun* cmp)
11  {
12      size_t i, j;
13      char* S = (char*)array;
14      char* temp = (char*)malloc(sizeof(char) * size);
15      i = n - 1;
16      while (i-- > 0)
17      {
18          memcpy(temp, S + i * size, size);
19          j = i;
20          while (++j < n && (*cmp)(temp, S + j * size) > 0);
21          if (--j == i) continue;
22          memcpy(S + i * size, S + (i + 1) * size, size * (j - i));
23          memcpy(S + j * size, temp, size);
24      }
25      free(temp);
26  }
```

qsort_median_insert 함수의 남은 input size 가 임계값보다 작아졌을 경우 실행하는 삽입 정렬 함수이다.

6) qsort_median_insert 함수

```
120  void qsort_median_insert(void* array, size_t n, size_t size, _Cmpfun* cmp)
121  {
122      if (n < 30)
123      {
124          insertion_sort((ELEMENT*)array, n);
125      }
126      else
127      {
128          while (1 < n)
129          {
130              size_t  i = 0;
131              size_t  j = n - 1;
132              size_t  r = median(i, j);
133              char* qi = (char*)array;
134              char* qj = qi + size * j;
135              char* qr = qi + size * r;
136              {
137                  char    buf[MAX_BUF];
138                  char* q1 = qr;
139                  char* q2 = qj;
140                  size_t  m, ms;
141
142                  for (ms = size; 0 < ms; ms -= m, q1 += m, q2 += m) {
143                      m = (ms < sizeof(buf)) ? ms : sizeof(buf);
144                      memcpy(buf, q1, m);
145                      memcpy(q1, q2, m);
146                      memcpy(q2, buf, m);
147                  }
148              }
149              char* qp = qj;
```

우선 남은 input size 가 임계값 이하인지 확인하는 조건문으로 시작된다. 만약 조건에 해당할 경우 남은 부분에 대해서는 insertion sort 함수가 실행된다.

그렇지 않은 경우 quick sort 부분으로 넘어간다. 위의 qsort_orig 함수와 다른 점은 median 함수를 이용하여 임의의 pivot 값을 정한다는 것이다. 이 역시 pivot 값의 위치를 정한 후 해당 위치의 값과 가장 오른쪽 값을 바꾸어 정렬을 시작한다. 이를 통해 median 함수로 선택된 값을 pivot 으로 사용하는 것과 동일한 효과를 얻는다.

```
150
151             while (i < j)
152             {
153                 while (i < j && (*cmp)(qi, qp) <= 0)
154                     ++i, qi += size;
155                 while (i < j && (*cmp)(qp, qj) <= 0)
156                     --j, qj -= size;
157
158                 if (i < j) {
159                     char    buf[MAX_BUF];
160                     char* q1 = qi;
161                     char* q2 = qj;
162                     size_t  m, ms;
163
164                     for (ms = size; 0 < ms; ms -= m, q1 += m, q2 += m) {
165                         m = (ms < sizeof(buf)) ? ms : sizeof(buf);
166                         memcpy(buf, q1, m);
167                         memcpy(q1, q2, m);
168                         memcpy(q2, buf, m);
169                     }
170                     ++i, qi += size;
171
172                 }
173             }
174             if (qi != qp) {
175                 char    buf[MAX_BUF];
176                 char* q1 = qi;
177                 char* q2 = qp;
178                 size_t  m, ms;
179
180                 for (ms = size; 0 < ms; ms -= m, q1 += m, q2 += m) {
181                     m = (ms < sizeof(buf)) ? ms : sizeof(buf);
182                     memcpy(buf, q1, m);
183                     memcpy(q1, q2, m);
184                     memcpy(q2, buf, m);
185                 }
186             }
```

```
            }
        }
        j = n - i - 1, qi += size;
        if (j < i) {
            if (1 < j)
                qsort_median_insert(qi, j, size, cmp);
            n = i;
        }
        else {
            if (1 < i)
                qsort_median_insert(array, i, size, cmp);
            array = qi;
            n = j;
        }
    }
```

이하 부분은 qsort_orig 와 동일하나 재귀적으로 불러주는 함수가 qsort_median_insert 라는 점만 다르다.

7) qsort_median_insert_iter 함수

```
7    int flag = 0;
8    size_t oldn = 0;
```

추후 함수에서 사용되는 flag 와 oldn 변수이다.

```
203  void qsort_median_insert_iter(void* array, size_t n, size_t size, _Cmpfun* cmp)
204  {
205      if (flag == 1 && n > (oldn / 2))
206      {
207          insertion_sort((ELEMENT*)array, n);
208      }
209      else {
210
211          while (1 < n)
212          {
213
214              size_t  i = 0;
215              size_t  j = n - 1;
216              size_t  r = median(i, j);
217              char* qi = (char*)array;
218              char* qj = qi + size * j;
219              char* qr = qi + size * r;
220              {
221                  char    buf[MAX_BUF];
222                  char* q1 = qr;
223                  char* q2 = qj;
224                  size_t  m, ms;
225
226                  for (ms = size; 0 < ms; ms -= m, q1 += m, q2 += m) {
227                      m = (ms < sizeof(buf)) ? ms : sizeof(buf);
228                      memcpy(buf, q1, m);
229                      memcpy(q1, q2, m);
230                      memcpy(q2, buf, m);
231                  }
232              }
233              char* qp = qj;
```

우선 flag 값을 확인한다. 함수의 첫 번째 동작은 qsort 가 이루어져야 하기 때문에 한 번의 동작 이후 flag 값을 1 로 만든다. 이후 이전 동작의 n 값의 절반에 해당하는 oldn / 2 와 현재의 n 값을 비교해 더 큰 부분일 경우 insertion_sort 를 통한 비재귀 방식의 정렬을 시행하도록 하였다.

그 이외의 경우인 quick_sort 부분의 경우 임계값의 존재를 제외한다면 qsort_median_insert 함수와 동일하다.

```
235             while (i < j)
236             {
237                 while (i < j && (*cmp)(qi, qp) <= 0)
238                     ++i, qi += size;
239                 while (i < j && (*cmp)(qp, qj) <= 0)
240                     --j, qj -= size;
241
242                 if (i < j) {
243                     char    buf[MAX_BUF];
244                     char* q1 = qi;
245                     char* q2 = qj;
246                     size_t  m, ms;
247
248                     for (ms = size; 0 < ms; ms -= m, q1 += m, q2 += m) {
249                         m = (ms < sizeof(buf)) ? ms : sizeof(buf);
250                         memcpy(buf, q1, m);
251                         memcpy(q1, q2, m);
252                         memcpy(q2, buf, m);
253                     }
254                     ++i, qi += size;
255                 }
256
257             }
258
259             if (qi != qp) {
260                 char    buf[MAX_BUF];
261                 char* q1 = qi;
262                 char* q2 = qp;
263                 size_t  m, ms;
264
265                 for (ms = size; 0 < ms; ms -= m, q1 += m, q2 += m) {
266                     m = (ms < sizeof(buf)) ? ms : sizeof(buf);
267                     memcpy(buf, q1, m);
268                     memcpy(q1, q2, m);
269                     memcpy(q2, buf, m);
270                 }
271             }
```

qsort_median_insert 함수와 동일하다.

```
272
273                 j = n - i - 1, qi += size;
274                 if (j < i) {
275                     if (1 < j)
276                         qsort_median_insert_iter(qi, j, size, cmp);
277
278                     n = i;
279
280                 }
281                 else {
282                     if (1 < i)
283                         qsort_median_insert_iter(array, i, size, cmp);
284                     array = qi;
285
286                     n = j;
287                 }
288
289
290
291             }
292             flag = 1; oldn = n;
293         }
294
295     }
```

qsort_orig 와 동일하나 재귀적으로 호출하는 함수가 qsort_median_insert_iter 라는 점이 다르며, while loof 밖에 flag 를 1 로 설정하는 부분과 현재의 n 값을 oldn 에 저장하여 다음 함수 실행 시 사용될 수 있도록 한다.

3. 실행 결과

각 함수(1 번, 21 번, 22 번, 23 번)에 대해서 5 가지의 input size(1024, 2048, 4096, 32768, 131072)를 준비하였고, 각각 5 회씩 실행한 결과의 평균을 구했다.

1) 1 번 함수(stdlib.h 의 qsort)

(1) input size : 1024

평균 : 1.547ms

(2) input size : 2048

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 2.702ms
```

평균 : 3.4682ms

(3) input size : 4096

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 10.037ms
```

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 11.460ms
```

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 6.366ms
```

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 8.238ms
```

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 7.588ms
```

평균 : 8.7378ms

(4) input size : 32768

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 80.285ms
```

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 149.018ms
```

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 88.843ms
```

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 83.679ms
```

```
<Function Number = 1>
 ELEMENT type of size 32: Time taken = 86.567ms
```

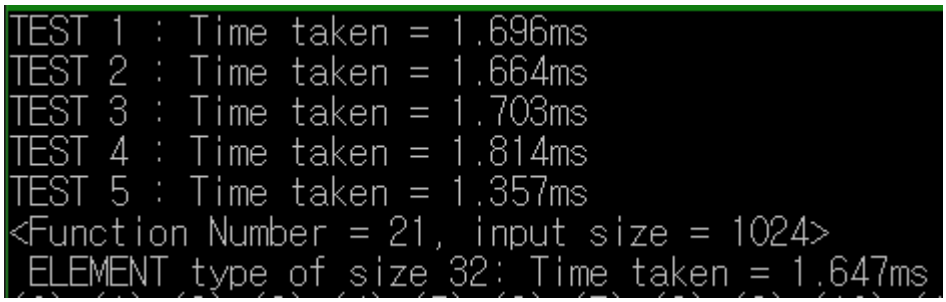
평균 : 97.6784ms

(5) input size : 131072

평균 : 333.2322ms

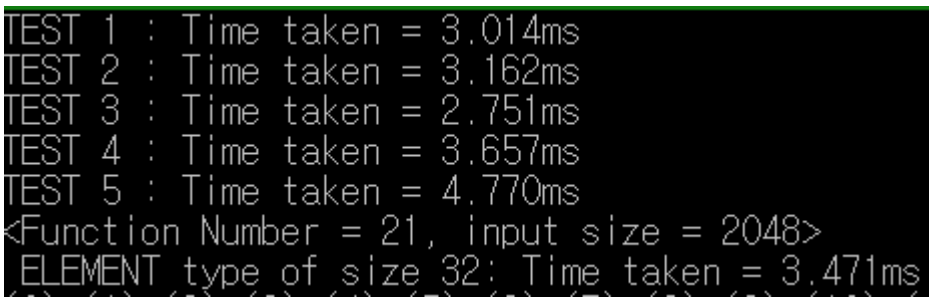
2) 21 번 함수(qsort_orig)

(1) input size : 1024



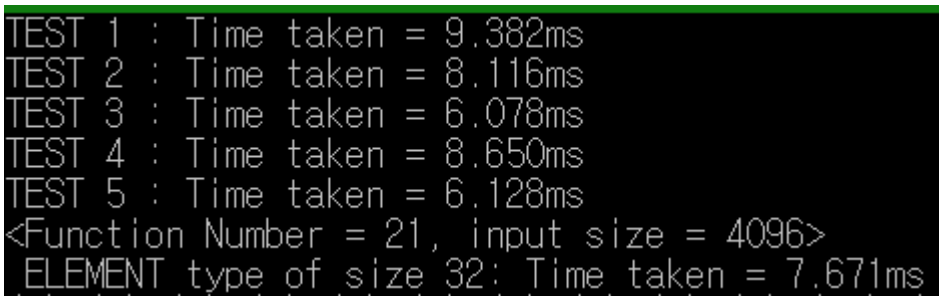
평균 : 1.647ms

(2) input size : 2048



평균 : 3.471ms

(3) input size : 4096



평균 : 7.671ms

(4) input size : 32768

```
TEST 1 : Time taken = 92.063ms
TEST 2 : Time taken = 96.435ms
TEST 3 : Time taken = 79.860ms
TEST 4 : Time taken = 79.738ms
TEST 5 : Time taken = 98.519ms
<Function Number = 21, input size = 32768>
 ELEMENT type of size 32: Time taken = 89.323ms
```

평균 : 89.323ms

(5) input size : 131072

```
TEST 1 : Time taken = 254.649ms
TEST 2 : Time taken = 273.073ms
TEST 3 : Time taken = 262.814ms
TEST 4 : Time taken = 273.710ms
TEST 5 : Time taken = 292.738ms
<Function Number = 21, input size = 131072>
 ELEMENT type of size 32: Time taken = 271.397ms
```

평균 : 271.397ms

3) 22 번 함수(qsort_median_insert)

(1) input size : 1024

```
TEST 1 : Time taken = 1.044ms
TEST 2 : Time taken = 1.048ms
TEST 3 : Time taken = 1.166ms
TEST 4 : Time taken = 1.123ms
TEST 5 : Time taken = 1.235ms
<Function Number = 22, input size = 1024>
 ELEMENT type of size 32: Time taken = 1.123ms
```

평균 : 1.123ms

(2) input size : 2048

```
TEST 1 : Time taken = 2.574ms
TEST 2 : Time taken = 2.687ms
TEST 3 : Time taken = 2.262ms
TEST 4 : Time taken = 2.628ms
TEST 5 : Time taken = 2.534ms
<Function Number = 22, input size = 2048>
 ELEMENT type of size 32: Time taken = 2.537ms
```

평균 : 2.537ms

(3) input size : 4096

```
TEST 1 : Time taken = 6.422ms
TEST 2 : Time taken = 8.583ms
TEST 3 : Time taken = 7.575ms
TEST 4 : Time taken = 8.146ms
TEST 5 : Time taken = 7.325ms
<Function Number = 22, input size = 4096>
 ELEMENT type of size 32: Time taken = 7.610ms
```

평균 : 7.610ms

(4) input size : 32769

```
TEST 1 : Time taken = 55.678ms
TEST 2 : Time taken = 71.285ms
TEST 3 : Time taken = 56.345ms
TEST 4 : Time taken = 63.725ms
TEST 5 : Time taken = 46.337ms
<Function Number = 22, input size = 32768>
 ELEMENT type of size 32: Time taken = 58.674ms
```

평균 : 58.674ms

(5) input size : 131072

```
TEST 1 : Time taken = 275.028ms
TEST 2 : Time taken = 238.935ms
TEST 3 : Time taken = 250.460ms
TEST 4 : Time taken = 206.792ms
TEST 5 : Time taken = 248.939ms
<Function Number = 22, input size = 131072>
 ELEMENT type of size 32: Time taken = 244.031ms
```

평균 : 244.031ms

4) 23 번 함수(qsort_median_insert_iter)

(1) input size : 1024

```
TEST 1 : Time taken = 3.972ms
TEST 2 : Time taken = 3.582ms
TEST 3 : Time taken = 4.790ms
TEST 4 : Time taken = 4.243ms
TEST 5 : Time taken = 3.992ms
<Function Number = 23, input size = 1024>
 ELEMENT type of size 32: Time taken = 4.116ms
```

평균 : 4.116ms

(2) input size : 2048

```
TEST 1 : Time taken = 5.404ms
TEST 2 : Time taken = 6.484ms
TEST 3 : Time taken = 5.370ms
TEST 4 : Time taken = 5.437ms
TEST 5 : Time taken = 5.204ms
<Function Number = 23, input size = 2048>
 ELEMENT type of size 32: Time taken = 5.580ms
```

평균 : 5.580ms

(3) input size : 4096

```
TEST 1 : Time taken = 23.862ms
TEST 2 : Time taken = 21.481ms
TEST 3 : Time taken = 28.258ms
TEST 4 : Time taken = 28.222ms
TEST 5 : Time taken = 30.893ms
<Function Number = 23, input size = 4096>
 ELEMENT type of size 32: Time taken = 26.543ms
```

평균 : 26.543ms

(4) input size : 32768

```
TEST 1 : Time taken = 2625.009ms
TEST 2 : Time taken = 2387.343ms
TEST 3 : Time taken = 1354.983ms
TEST 4 : Time taken = 1156.938ms
TEST 5 : Time taken = 1639.492ms
<Function Number = 23, input size = 32768>
 ELEMENT type of size 32: Time taken = 1832.753ms
```

평균 : 1832.753ms

(5) input size : 131072

```
TEST 1 : Time taken = 22078.154ms
TEST 2 : Time taken = 19927.021ms
TEST 3 : Time taken = 19422.592ms
TEST 4 : Time taken = 22815.262ms
TEST 5 : Time taken = 10850.832ms
<Function Number = 23, input size = 131072>
 ELEMENT type of size 32: Time taken = 19018.771ms
```

평균 : 19018.771ms

4. 결과 분석 및 제언

(단위 : ms)	함수 1 번	함수 21 번	함수 22 번	함수 23 번
1024	1.547	1.647	1.123	4.116
2048	3.4682	3.471	2.537	5.580
4096	8.7378	7.671	7.610	26.543
32768	97.6784	89.323	58.674	1832.753
131072	333.2322	271.397	244.031	19018.771

본 프로그램의 함수 별 입력 크기에 따른 실행 시간을 표로 정리해보았다.

가장 먼저 눈에 띄는 것은 23 번 함수가 1 번~22 번 함수에 비해서 확연하게 느린 성능을 보여주고 있다는 것이다. 그 원인은 비교적 시간복잡도가 높은 삽입정렬(Avg case : O(n^2))을 크기가 큰 부분에 대해서 실행하였기 때문이라고 생각한다.

다음으로 1 번 함수와 21 번 함수는 그 실행 속도에서 큰 차이를 보이지 않았다. 오히려 큰 input size 에 대해서 근소하게 빠른 모습을 보였는데, 이를 통해 pivot 을 왼쪽 끝에서 선정한 후 위치를 오른쪽 끝으로 바꾸는 동작이 실행 속도에는 큰 영향을 주지 않았음을 알 수 있었다.

그러나 22 번 함수와 1 번 함수는 큰 입력 크기에 대해서 입력 속도에서 유의미한 차이를 보였다. 이는 작은 n 에 대한 insertion sort 선택과 단순히 왼쪽 원소를 pivot 으로 택하는 것이 아닌 무작위적으로 pivot 을 선정함으로써 worst case 를 피할 확률이 높아졌기 때문이라고 볼 수 있다.

또한 22 번 함수와 23 번 함수를 비교하였을 때 22 번 함수는 큰 부분에 대해서 재귀를 시행한 후 입력 크기가 충분히 작아졌을 때 반복형 함수를 동작한 반면 23 번 함수는 반대로 큰 부분에 대해서는 반복으로, 작은 부분에는 재귀로 접근하였다. 그 결과 22 번 함수에 비해 23 번 함수의 실행 속도가 느린 것으로 나타났다. 이를 통해 작은 input size 에 대해 재귀를 실행하는 것이 무조건적인 성능 향상을 가져다 주는 것이 아님을 알 수 있었다.