Exercise 1

```
postgres=# \d table1
                 Table "public.table1"
  Column  |     Type      | Collation | Nullable | Default
----------+---------------+-----------+----------+---------
 sorted   | integer       |           |          |
 unsorted | integer       |           |          |
 rndm     | integer       |           |          |
 dummy    | character(40) |           |          |
Indexes:
    "index_sorted" btree (sorted)
    "index_unsorted" btree (unsorted)
```

```
create index index_sorted on table1(sorted);
create index index_unsorted on table1(unsorted);
```

- `sorted`의 경우 정렬되어 있으므로 clustered index
- `unsorted`의 경우 정렬되어 있지 않으므로 non-clustered index

Exercise 2

a

seq scan

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..156186.33 rows=100 width=4) (actual time=142.377..2649.640 rows=114 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on table1  (cost=0.00..155176.33 rows=42 width=4) (actual time=71.036..2644.305 rows=38 loops=3)
         Filter: (rndm = 1005)
         Rows Removed by Filter: 3333295
 Planning Time: 2.692 ms
 Execution Time: 2649.754 ms
(8 rows)
```

```
explain analyze select rndm from table1 where rndm = 1005;
```

index scan

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Index Scan using index_sorted on table1  (cost=0.43..8.68 rows=14 width=8) (actual time=1.552..1.555 rows=5 loops=1)
   Index Cond: (sorted = 1999231)
 Planning Time: 0.331 ms
 Execution Time: 1.586 ms
(4 rows)
```

```
explain analyze select sorted, rndm from table1 where sorted = 1999231;
```

index only scan

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Index Only Scan using index_sorted on table1  (cost=0.43..4.68 rows=14 width=4) (actual time=1.939..1.943 rows=5 loops=1)
   Index Cond: (sorted = 1999231)
   Heap Fetches: 0
 Planning Time: 0.157 ms
 Execution Time: 1.972 ms
(5 rows)
```

```
explain analyze select sorted from table1 where sorted = 1999231;
```

b

clustered index

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Index Only Scan using index_sorted on table1  (cost=0.43..95.68 rows=4071 width=4) (actual time=0.015..0.971 rows=3840 loops=1)
   Index Cond: (sorted > 1999231)
   Heap Fetches: 0
 Planning Time: 1.290 ms
 Execution Time: 1.219 ms
(5 rows)
```

```
explain analyze select sorted from table1 where sorted > 1999231;
```

non-clustered index

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Index Only Scan using index_unsorted on table1  (cost=0.43..91.08 rows=3808 width=4) (actual time=0.213..1.258 rows=3765 loops=1)
   Index Cond: (unsorted > 1999231)
   Heap Fetches: 0
 Planning Time: 2.308 ms
 Execution Time: 1.471 ms
(5 rows)
```

```
explain analyze select unsorted from table1 where unsorted > 1999231;
```

시간 비교

clustered index의 경우 1999231을 찾은 다음 relation에서 linear scan을 하면 되지만 non-clustered index의 경우 계속해서 B-tree를 탐색하면서 레코드에 대한 포인터를 가져와야 하므로 시간이 상대적으로 더 소요된다.

C

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using index_sorted on table1  (cost=0.43..150.86 rows=1 width=8) (actual time=1.027..1.028 rows=0 loops=1)
   Index Cond: (sorted > 1999231)
   Filter: (rndm = 1005)
   Rows Removed by Filter: 3840
 Planning Time: 0.216 ms
 Execution Time: 1.054 ms
(6 rows)
```

```
explain analyze SELECT sorted, rndm FROM table1 WHERE sorted>1999231 AND
rndm=1005;
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..166603.00 rows=100 width=8) (actual time=26.347..2501.316 rows=114 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on table1  (cost=0.00..165593.00 rows=42 width=8) (actual time=69.360..2497.327 rows=38 loops=3)
         Filter: ((sorted < 1999231) AND (rndm = 1005))
         Rows Removed by Filter: 3333295
 Planning Time: 0.180 ms
 Execution Time: 2501.352 ms
(8 rows)
```

```
explain analyze SELECT sorted, rndm FROM table1 WHERE sorted<1999231 AND
rndm=1005;
```

```
postgres=# select sorted from table1 where sorted = (select max(sorted) from table1);
 sorted
---------
 1999999
 1999999
 1999999
 1999999
 1999999
(5 rows)
```

```
select sorted from table1 where sorted = (select max(sorted) from table1);
```

query plan 다른 이유

`sorted` 컬럼의 최댓값이 1999999인 것을 볼 수 있다. 첫 번째 쿼리의 경우 index를 통한 레코드를 찾은 이후 linear 탐색해야 되는 범위가 매우 적지만 두번째 쿼리의 경우 index를 이용해 레코드를 찾은 이후 linear 탐색해야 되는 범위가 매우 넓

다. 따라서 seq scan과 비교하였을 때 큰 효과가 없으며 오히려 B-tree 탐색을 위한 오버헤드 발생으로 효율이 더 떨어질 수도 있다.

Exercise 3

inserting first

```
postgres=# insert into table10 (select * from pool);
INSERT 0 5000000
Time: 6078.649 ms (00:06.079)
postgres=# create index tmp on table10(val);
CREATE INDEX
Time: 2166.058 ms (00:02.166)
```

```
create table table10 (val integer);
insert into table10 (select * from pool);
create index tmp on table10(val);
```

- 소요시간: 8244.707 ms

creating index first

```
postgres=# create index tmp2 on table20(val);
CREATE INDEX
Time: 5.250 ms
postgres=# insert into table20 (select * from pool);
INSERT 0 5000000
Time: 12052.946 ms (00:12.053)
```

```
create table table20 (val integer);
create index tmp2 on table20(val);
insert into table20 (select * from pool);
```

- 소요시간: 12058.196 ms

시간이 차이 나는 이유

insert 이후에 index를 생성하면 정렬되어 있는 컬럼에서 bottom-up 방식으로 B-tree를 빠르게 생성할 수 있다. 하지만 index를 생성한 이후 insert를 하게 되면 레코드가 삽입될 때마다 B-tree를 탐색해야 되고 노드의 크기가 일정 수준 넘어가게 되면 split 연산까지 추가적으로 발생한다. 이와 같은 이유로 insert를 먼저 하는 것이 효율적이다.