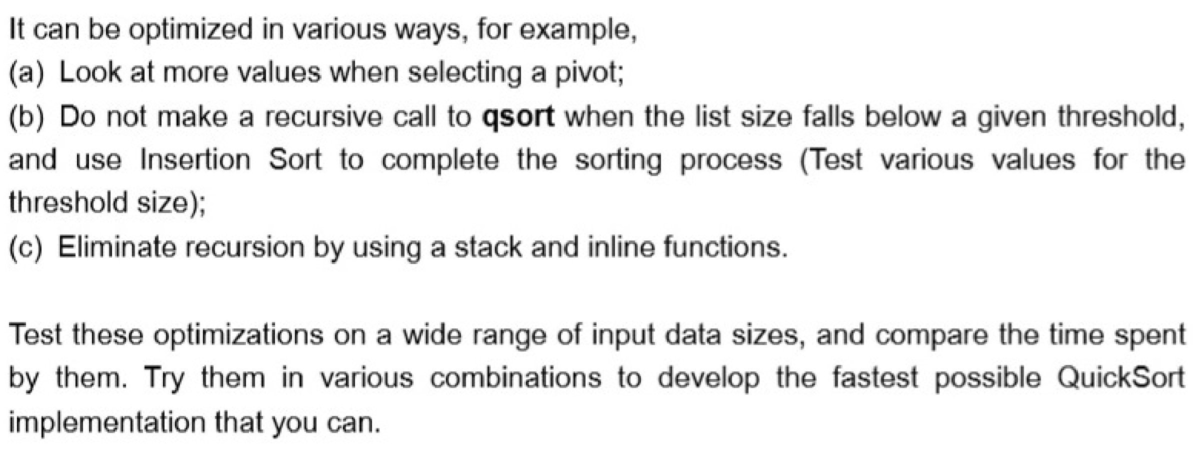
Report for lab-3

- name: He Xiang (贺翔)
- student ID: 2017141461079
- date: 2018/12/26
- Experimental environment

> OS: Windows 10
>
> IDE: Visual Studio 2017 & vim 8.1 & VS code

Task-1

- Requirement



It can be optimized in various ways, for example,
(a) Look at more values when selecting a pivot;
(b) Do not make a recursive call to **qsort** when the list size falls below a given threshold, and use Insertion Sort to complete the sorting process (Test various values for the threshold size);
(c) Eliminate recursion by using a stack and inline functions.

Test these optimizations on a wide range of input data sizes, and compare the time spent by them. Try them in various combinations to develop the fastest possible QuickSort implementation that you can.

- Algorithm

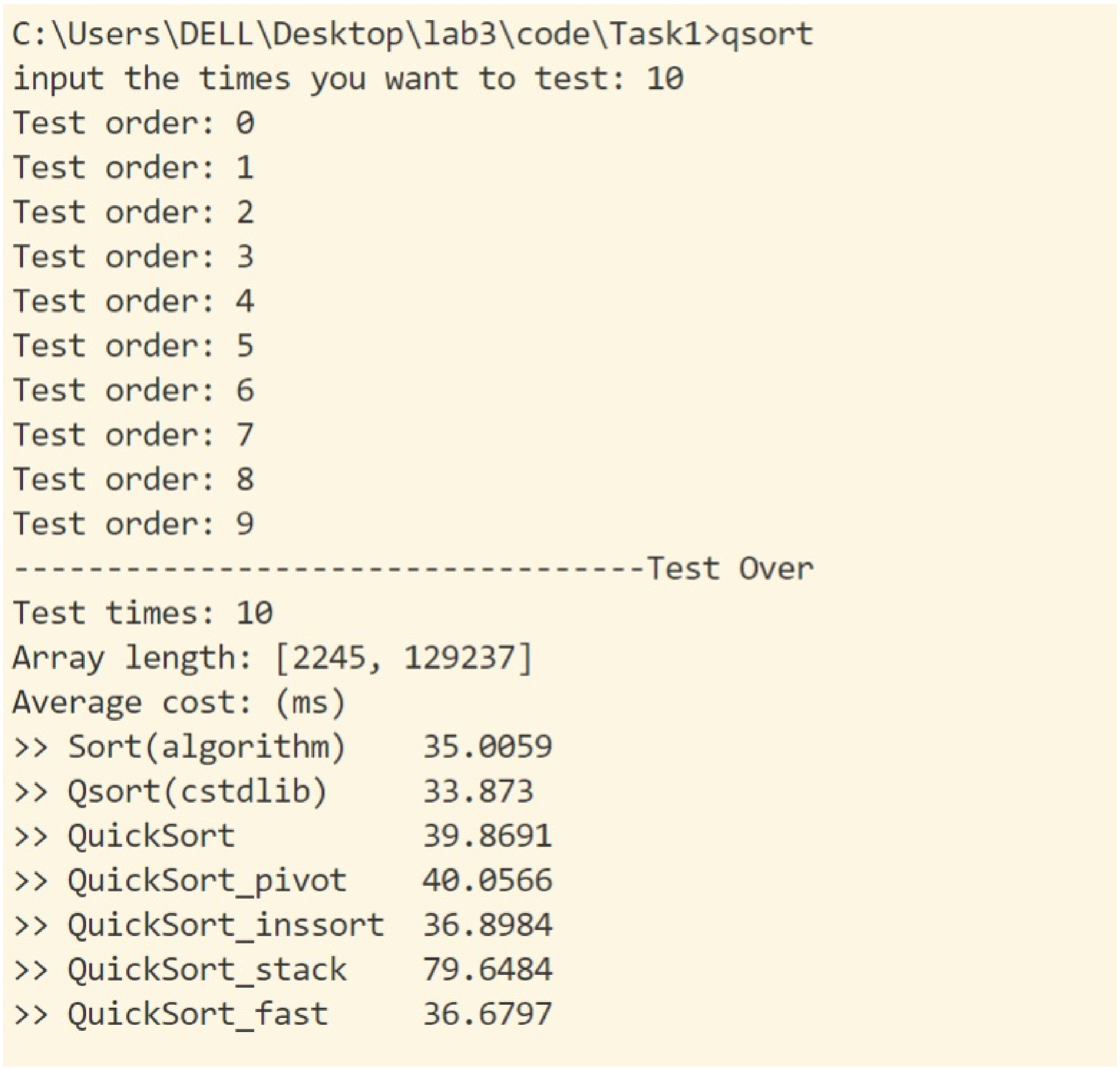
For the last quicksort algorithm, I choose use (a) and (b) to make the sort faster.

Because, in condition (c), the sort will be more slower.

  - (a): choose the middle one between A[l], A[r] and A[pivot].
  - (b): set the threshold as 8.
  - (c): use stack to keep each subarray's range, and implement the sort by loop.
- Source code files

> ./code/Task1/quickSort.h # implement those sort algorithm
>
> ./code/Task1/qsort.cc # with main function and do some test.

- Experimental results



```
C:\Users\DELL\Desktop\lab3\code\Task1>qsort
input the times you want to test: 10
Test order: 0
Test order: 1
Test order: 2
Test order: 3
Test order: 4
Test order: 5
Test order: 6
Test order: 7
Test order: 8
Test order: 9
----------------------------------Test Over
Test times: 10
Array length: [2245, 129237]
Average cost: (ms)
>> Sort(algorithm)    35.0059
>> Qsort(cstdlib)     33.873
>> QuickSort          39.8691
>> QuickSort_pivot    40.0566
>> QuickSort_inssort  36.8984
>> QuickSort_stack    79.6484
>> QuickSort_fast     36.6797
```

It shows that, those 7 kinds of quicksort algorithm will be test 10 times.

And the length of the array be sorted will be in range 2245 and 129237, all for random.

1. Sort(algorithm) is the sort() function in namespace std, "algorithm".
2. Qsort(cstdlib) is the qsort() function in namespace std, "cstdlib".
3. QuickSort is the original function the Task-1 give.
4. QuickSort_pivot is the sort algorithm with implement of (a).
5. QuickSort_inssort is the sort algorithm with implement of (b).
6. QuickSort_stack is the sort algorithm with implement of (c).
7. QuickSort_fast is the sort algorithm with implement of combine of (a) and (b).

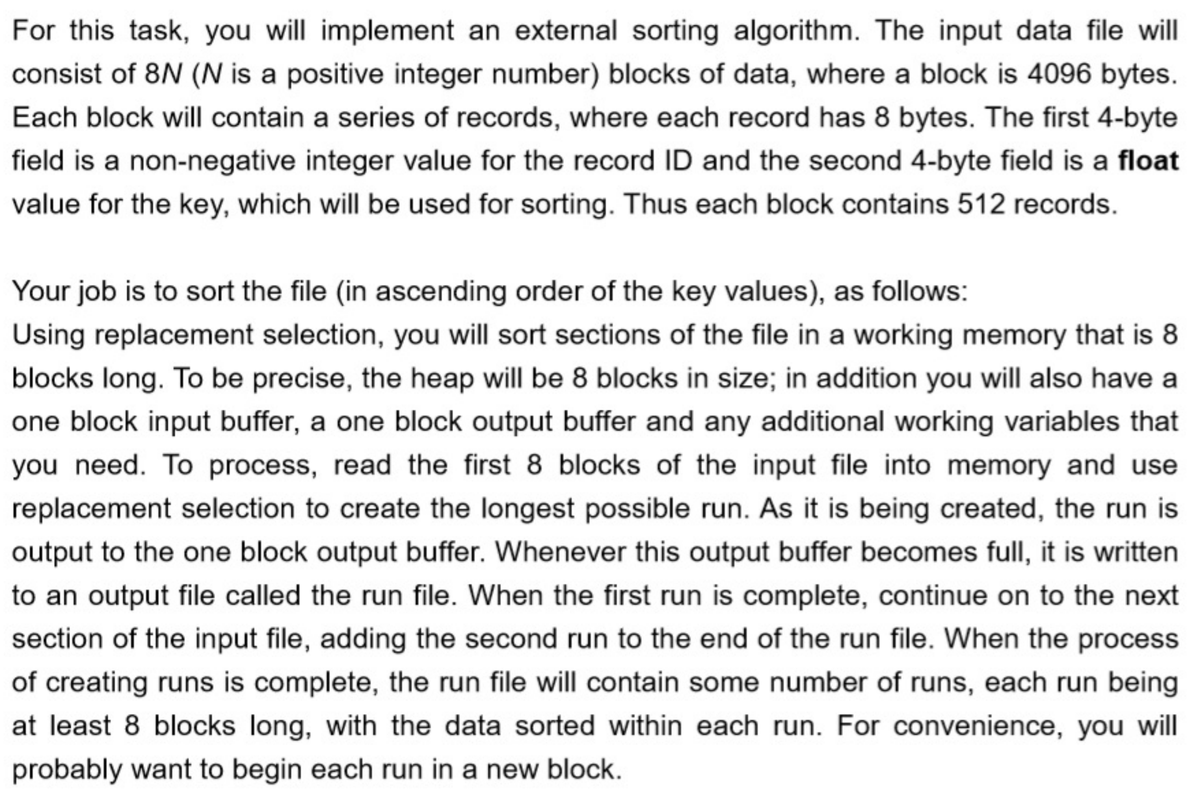
- Analysis and discussion

  The quicksort algorithm average take time efficiency for $O(N \cdot \log N)$, but for some optimization, the algorithm will be faster.

  for cases (a) and (b), it will make qsort faster. But for case (c), the operations like pop and push for stack take much time, so it will be slower.

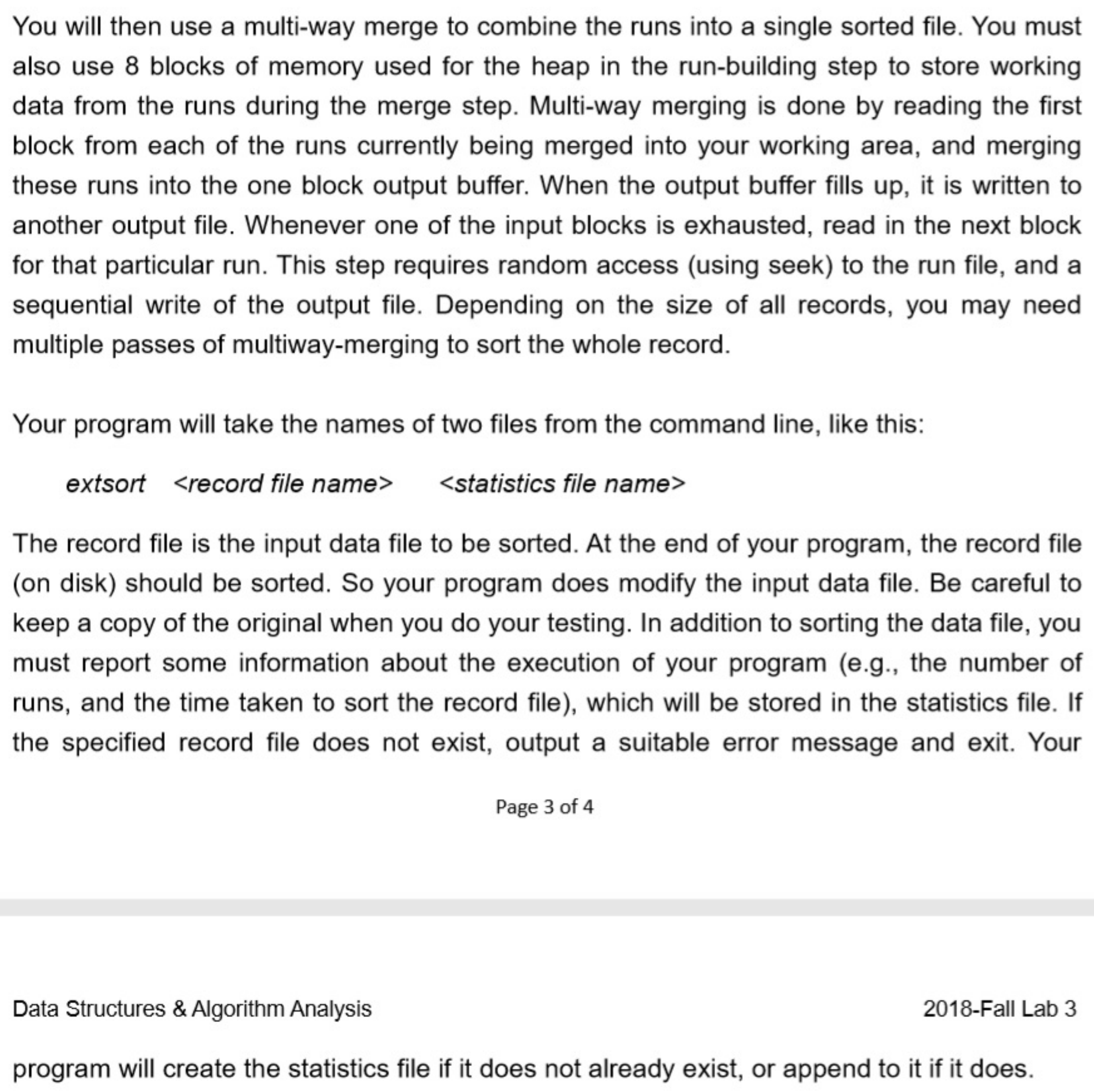
Task-2

- Requirement



For this task, you will implement an external sorting algorithm. The input data file will consist of 8*N* (*N* is a positive integer number) blocks of data, where a block is 4096 bytes. Each block will contain a series of records, where each record has 8 bytes. The first 4-byte field is a non-negative integer value for the record ID and the second 4-byte field is a **float** value for the key, which will be used for sorting. Thus each block contains 512 records.

Your job is to sort the file (in ascending order of the key values), as follows:
Using replacement selection, you will sort sections of the file in a working memory that is 8 blocks long. To be precise, the heap will be 8 blocks in size; in addition you will also have a one block input buffer, a one block output buffer and any additional working variables that you need. To process, read the first 8 blocks of the input file into memory and use replacement selection to create the longest possible run. As it is being created, the run is output to the one block output buffer. Whenever this output buffer becomes full, it is written to an output file called the run file. When the first run is complete, continue on to the next section of the input file, adding the second run to the end of the run file. When the process of creating runs is complete, the run file will contain some number of runs, each run being at least 8 blocks long, with the data sorted within each run. For convenience, you will probably want to begin each run in a new block.


You will then use a multi-way merge to combine the runs into a single sorted file. You must also use 8 blocks of memory used for the heap in the run-building step to store working data from the runs during the merge step. Multi-way merging is done by reading the first block from each of the runs currently being merged into your working area, and merging these runs into the one block output buffer. When the output buffer fills up, it is written to another output file. Whenever one of the input blocks is exhausted, read in the next block for that particular run. This step requires random access (using seek) to the run file, and a sequential write of the output file. Depending on the size of all records, you may need multiple passes of multiway-merging to sort the whole record.

Your program will take the names of two files from the command line, like this:

extsort <record file name> <statistics file name>

The record file is the input data file to be sorted. At the end of your program, the record file (on disk) should be sorted. So your program does modify the input data file. Be careful to keep a copy of the original when you do your testing. In addition to sorting the data file, you must report some information about the execution of your program (e.g., the number of runs, and the time taken to sort the record file), which will be stored in the statistics file. If the specified record file does not exist, output a suitable error message and exit. Your

Page 3 of 4

Data Structures & Algorithm Analysis 2018-Fall Lab 3

program will create the statistics file if it does not already exist, or append to it if it does.

- Algorithm

1. relevant documents

   > REC.dat # original records file
   >
   > STAT.dat # the run file information records.
   >
   > RUNFILE.tmp # the temporary file for runs.
   >
   > cp_REC.dat # the copy of REC.dat

2. other variables

   > `vetor<pair<int, int> >` to store the index of each run in RUNFILE.tmp

3. `std::rename(REC.dat, cp_REC.dat);`.
4. `ifstream(cp_REC.dat, ios::binary); ofstream(RUNFILE.tmp, ios::binary)`. Use replacement select to build run file: RUNFILE.tmp
5. `ifstream(RUNFILE.tmp, ios::binary); ofstream(REC.dat, ios::binary);`. Merge runs in RUNFILE.tmp by 8-way merge.

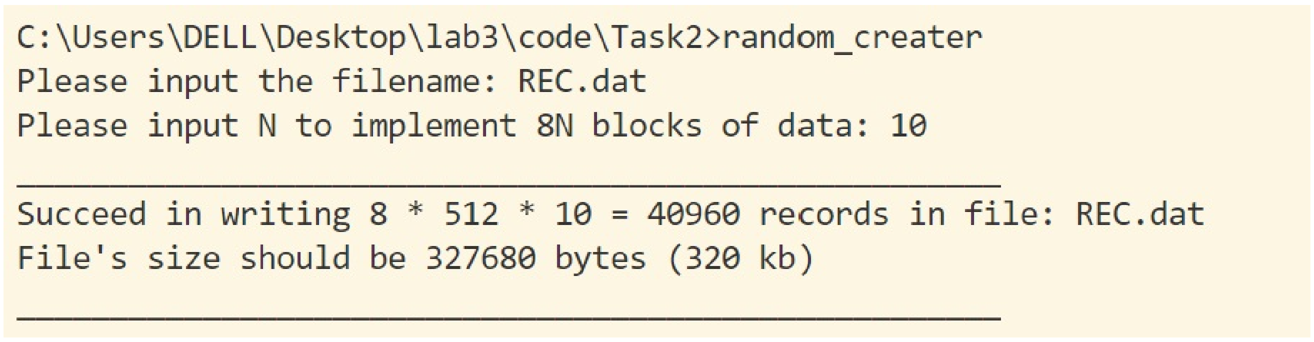
6. assume the original runs' number N. We need merge times M be $\lceil \log_8 N \rceil$. If M > 1, repeat step 5 while filename changed.
7. when the all merges are done, if M is odd, `remove(RUNFILE.tmp)`, if M is even, `remove(REC.dat); rename(RUNFILE.dat, REC.dat)`.
8. all run file information records written in STAT.dat.

- Source code files

> ./code/Task2/random_creater.cc # create random records file for this task.
>
> ./code/Task2/check_sorted.cc # check whether the file is sorted.
>
> ./code/Task2/extsort_init.h # class's declare
>
> ./code/Task2/extsort_init.cc # class's implementation
>
> ./code/Task2/extsort.cc # main function, for test

- Experimental results

- **create new records file**



```
C:\Users\DELL\Desktop\lab3\code\Task2>random_creater
Please input the filename: REC.dat
Please input N to implement 8N blocks of data: 10
_____________________________________________________
Succeed in writing 8 * 512 * 10 = 40960 records in file: REC.dat
File's size should be 327680 bytes (320 kb)
_____________________________________________________
```

- **sort the file REC.dat and store runs's information in STAT.dat**

```
C:\Users\DELL\Desktop\lab3\code\Task2>extsort REC.dat STAT.dat
start building runfile...
run-build progress: 17.1558%
run-build progress: 36.8384%
run-build progress: 56.9189%
run-build progress: 76.8872%
run-build progress: 95.6885%
run-build progress: 100%
runfile has been built

start merging...
need merge times: 1
>> merge_order: 0
done


___________________
extsort success!
```

- check whether the file REC.dat is sorted, cp_REC.dat is the copy of original REC.dat

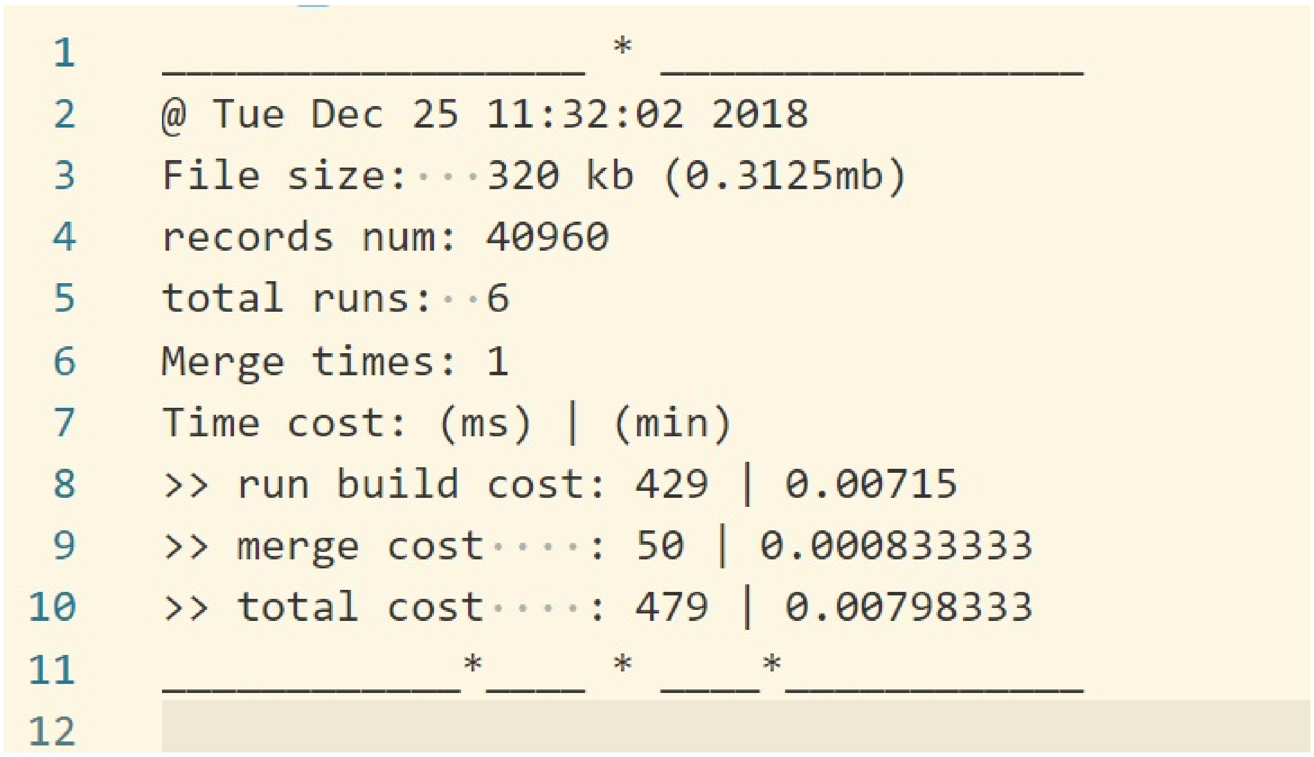
```
C:\Users\DELL\Desktop\lab3\code\Task2>check_sorted REC.dat

__________________________________
file: REC.dat has been sorted
__________________________________

C:\Users\DELL\Desktop\lab3\code\Task2>check_sorted cp_REC.dat

__________________________________
file: cp_REC.dat has not beed sorted
prev-idx: 0  key: 2600.27
curr-idx: 1  key: 2350.27
__________________________________
```

- For STAT.dat



```
_________________ * _________________
@ Tue Dec 25 11:32:02 2018
File size:   320 kb (0.3125mb)
records num: 40960
total runs:  6
Merge times: 1
Time cost: (ms) | (min)
>> run build cost: 429 | 0.00715
>> merge cost    : 50 | 0.000833333
>> total cost    : 479 | 0.00798333
____________*____ * ____*____________

```

We can see, for records file (size 320 kb, 40960 records), the external sorting algorithm will take 479 ms.

- Analysis and discussion

For this external sorting algorithm, we use Replacement select to build original run file, and use 8-way merge to merge each run. This will make sort fast.

But for we store all runs in one run file, we need the file pointer jump frequently wen we read the run file. So this will cost much time.

Task-3

- Requirement

> Implement the following dictionary ADT by means of a hash table with linear probing as the collision resolution policy. Your dictionary uses a non-negative integer as the key, and a string of characters as the associated record.
>
> Using empirical simulation, determine the cost of insert and delete as the load factor grows.
>
> Then, repeat the experiment using quadratic probing and pseudo-random probing, and compare the relative performance of the three collision resolution policies (i.e., linear probing, quadratic probing, and pseudo-random probing).

- Algorithm

1. linear probing

`__probe_(k, i) = i`

2. pseudo-random probing

```
int *perm;
perm = new int[size];
for (int i = 0; i < size - 1; i ++)
    perm[i] = rand() % size;

__probe__(k, i) = perm[i - 1];
```

3. quadratic probing

`__probe___(k, i) = (i * i + i) / 2`

- Source code files

> ./code/Task3/Dict.h # virtual base class
>
> ./code/Task3/HashTable.h # hashdict class implement
>
> ./code/Task3/Hash.cc # hash insert & delete test

- Experimental results

for test 10 rounds, each hashtable's maxsize is 1e4.

load factor	linear insert	perm insert	quad insert	-	linear delete	perm delete	quad delete
0.100	1.250	1.500	1.250	-	1.000	1.002	1.031
0.200	1.750	1.750	1.750	-	1.000	1.250	1.000
0.300	1.008	1.508	1.008	-	1.000	1.000	1.563
0.400	1.201	1.686	1.205	-	1.000	1.127	1.008
0.500	1.910	2.527	1.965	-	1.563	2.547	1.063
0.600	3.084	3.375	2.512	-	1.545	1.500	1.041
0.700	2.977	3.480	2.689	-	5.355	1.375	1.844
0.800	108.238	7.850	20.475	-	1.633	1.156	1.000
0.900	518.873	3.834	15.521	-	1.145	6.008	3.113

value stands for average cost of query hash table.

- Analysis and discussion

As we can see, the pseudo-random probing is best in insert operation. And quadratic probing is more efficient in delete operation. The linear probing is the worst one.