Project 3: Cars on Campus

Author

2016-12-24

Contents

1 Introduction

1.1 Concept description

1.1.1 Sorting algorthm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order)
2. The output is a permutation (reordering) of the input.

Further, the data is often taken to be in an array, which allows random access, rather than a list, which only allows sequential access, though often algorithms can be applied with suitable modification to either type of data.
Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Comparison sorting algorithms have a fundamental requirement of $O(nlogn)$ comparisons (some input sequences will require a multiple of n log n comparisons); algorithms not based on comparisons, such as counting sort, can have better performance. Although many consider sorting a solved problem—asymptotically optimal algorithms have been known since the mid-20th century—useful new algorithms are still being invented, with the now widely used Timsort dating to 2002, and the library sort being first published in 2006.
Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures such as heaps and binary trees, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and upper and lower bounds.

1.1.2 Clarification

Usually we have a lot of attrbutions to describe a sort algorthm, and here are some clarfications.

1. **Computational complexity** (worst, average and best behavior) in terms of the size of the list (n). For typical serial sorting algorithms good behavior is O(n log n), with parallel sort in O(log2 n), and bad behavior is O(n2). (See Big O notation.) Ideal behavior for a serial sort is O(n), but this is not possible in the average case. Optimal parallel sorting is O(log n). Comparison-based sorting algorithms, need at least O(n log n) comparisons for most inputs.

2. **Computational complexity of swaps** (for "in-place" algorithms).

3. **Memory usage** (and use of other computer resources). In particular, some sorting algorithms are "in-place". Strictly, an in-place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(n)) additional memory is considered "in-place".

4. **Recursion**: Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).

5. **Stability**: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).

6. **Whether or not they are a comparison sort**: A comparison sort examines the data only by comparing two elements with a comparison operator.

7. **General method**: insertion, exchange, selection, merging, etc. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort. Also whether the algorithm is serial or parallel. The remainder of this discussion almost exclusively concentrates upon serial algorithms and assumes serial operation.

8. **Adaptability**: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

1.2 Project Description

1.2.1 Project Background

Zhejiang University has 8 campuses and a lot of gates. From each gate we can collect the in/out times and the plate numbers of the cars crossing the gate. Now with all the information available, you are supposed to tell, at any specific time point, the number of cars parking on campus, and at the end of the day find the cars that have parked for the longest time period.

1.2.2 Our tasks

Our tasks are:

- store the records and get rid of invaild records
- use a efficient sort way to sort the records in order of time and tell the number of cars at some specific time points
- find out the cars that have longest parking time

1.2.3 Input Spection

Each input file contains one test case. Each case starts with two positive integers N (10^4), the number of records, and K (8×10^4) the number of queries. Then N lines follow, each gives a record in the format:
plate_number $hh : mm : ss$ *status*
where *plate_number* is a string of 7 English capital letters or 1-digit numbers; $hh : mm : ss$ represents the time point in a day by *hour* : *minute* : *second*, with the earliest time being 00 : 00 : 00 and the latest 23 : 59 : 59; and *status* is either in or out.
Note that all times will be within a single day. Each in record is paired with the chronologically next record for the same car provided it is an *out* record. Any *in* records that are not paired with an *out* record are ignored, as are *out* records not paired with an *in* record. It is guaranteed that at least one car is well paired in the input, and no car is both *in* and *out* at the same moment. Times are recorded using a 24-hour clock.
Then K lines of queries follow, each gives a time point in the format $hh : mm : ss$. Note: the queries are given in **accending** order of the times.

1.2.4 Output Specification

For each query, output in a line the total number of cars parking on campus. The last line of output is supposed to give the plate number of the car that has parked for the longest time period, and the corresponding time length. If such a car is not unique, then output all of their plate numbers in a line in alphabetical order, separated by a space.

1.2.5 Sample Input

```
16 7
JH007BD 18:00:01 in
ZD00001 11:30:08 out
DB8888A 13:00:00 out
ZA3Q625 23:59:50 out
ZA133CH 10:23:00 in
ZD00001 04:09:59 in
JH007BD 05:09:59 in
ZA3Q625 11:42:01 out
JH007BD 05:10:33 in
ZA3Q625 06:30:50 in
JH007BD 12:23:42 out
ZA3Q625 23:55:00 in
JH007BD 12:24:23 out
ZA133CH 17:11:22 out
JH007BD 18:07:01 out
DB8888A 06:30:50 in
05:10:00
06:30:50
```

```
11:00:00
12:23:42
14:00:00
18:00:00
23:59:00
```

1.2.6 Sample Output

```
1
4
5
2
1
0
1
JH007BD ZD00001 07:20:09
```

2 Algorithm Specification

2.1 General

The most important algorithm in solving this problem is sort. While there are a large number of sorting algorithms, in practical implementations a few algorithms predominate. Insertion sort is widely used for small data sets, while for large data sets an asymptotically efficient sort is used, primarily heap sort, merge sort, or quicksort. Efficient implementations generally use a hybrid algorithm, combining an asymptotically efficient algorithm for the overall sort with insertion sort for small lists at the bottom of a recursion. Highly tuned implementations use more sophisticated variants, such as Timsort (merge sort, insertion sort, and additional logic), used in Android, Java, and Python, and introsort (quicksort and heap sort), used (in variant forms) in some C++ sort implementations and in .NET.
And in our program we use qsort. We believe that qsort will perform well in solving this problem.
qsort is a C standard library function that implements a polymorphic sorting algorithm for arrays of arbitrary objects according to a user-provided comparision function. It is named after the "quicker sort" algorithm, which was originally used to implement it in the Unix C library, although the C standard does not require it to implement quicksort.
The qsort in gcc, which is in stdlib/msort.c, is implemented in a rather complex way for efficiency. In other words, it is a hybrid algorithm. Basically it is a combination of merge sort, quick sort and insertion sort with quantities of other optimizations on CPU and memory for efficiency. Here is its routine.

1. caculate the object size, and use indirect sorting for large object sizes, that is, using pointers rather than objects itself.

2. caculate the extra needed space if using merge sort and try to allocate the memory. If the temporaray array is small, put it on the stack, otherwise

try to allocate it in the heap. If memory requirements are too high, then dont allcate memory and use _quicksort, which is in stdlib/qsort.c. It is implemented in a quick sort way, as it's name is.

(a) if allocate operation is successful, use merge sort.

i. If the size of each element is larger than 32 bytes, use indirect sort, otherwise use direct sort. And it does a lot of extra optimizations for max use of CPU.

ii. avoid allocating too much memory since this might have to be backed up by swap space.

(b) if allocate operation failed, use _quicksort, which is less efficient.

i. Select median value from among LO, MID, and HI. Rearrange LO and HI so the three values are sorted. This lowers the probability of picking a pathological pivot value and skips a comparison for both the left_pointer and right_pointer in the while loops.

ii. Also, it continues the sort in a iterative way. Using an explicit stack of pointer that store the next array partition to sort. To save time, this maximum amount of space required to store an array of SIZE_MAX is allocated on the stack. Assuming a 32-bit (64 bit) integer for size_t, this needs only 32 * sizeof(stack_node) == 256 bytes (for 64 bit: 1024 bytes). Pretty cheap, actually.

iii. Once the array is partially sorted by quicksort the rest is completely sorted using insertion sort, since this is efficient for partitions below a certain(MAX_THRESH) size.

2.1.1 Qsort Source Code

Here we give parts of qsort implementation code in gcc.

Code 1: part of qsort implementation

```
1  void
2  qsort (void *b, size_t n, size_t s, __compar_fn_t cmp)
3  {
4      return qsort_r (b, n, s, (__compar_d_fn_t) cmp, NULL);
5  }
6
7  void
8  qsort_r (void *b, size_t n, size_t s, __compar_d_fn_t cmp, void *arg)
9  {
10     size_t size = n * s;
11     char *tmp = NULL;
12     struct msort_param p;
13
14     /* For large object sizes use indirect sorting.  */
15     if (s > 32)
16     size = 2 * n * sizeof (void *) + s;
17
```

```
18
19      /*try to alloc/malloc memory, details  */
20
21      /* If the memory requirements are too high don't allocate memory.
    */
22      if (size / pagesize > (size_t) phys_pages)
23      {
24        _quicksort (b, n, s, cmp, arg);
25        return;
26      }
27
28      /* It's somewhat large, so malloc it.  */
29      int save = errno;
30      tmp = malloc (size);
31      __set_errno (save);
32      if (tmp == NULL)
33      {
34          /* Couldn't get space, so use the slower algorithm
35           that doesn't need a temporary array.  */
36          _quicksort (b, n, s, cmp, arg);
37          return;
38      }
39          p.t = tmp;
40      }
41
42     /*details omitted*/
43
44      if (s > 32)
45      {
46          /* Indirect sorting.  */
47          char *ip = (char *) b;
48          void **tp = (void **) (p.t + n * sizeof (void *));
49          void **t = tp;
50          void *tmp_storage = (void *) (tp + n);
51          while ((void *) t < tmp_storage)
52          {
53              *t++ = ip;
54              ip += s;
55          }
56
57          p.s = sizeof (void *);
58          p.var = 3;
59          msort_with_tmp (&p, p.t + n * sizeof (void *), n);
60
61          /* tp[0] .. tp[n - 1] is now sorted, copy around entries of
62          the original array.  Knuth vol. 3 (2nd ed.) exercise 5.2-10.
    */
63
64          /*details omitted*/
65
```

```
66          free (tmp);
67      }
68      /*details omitted*/
69  }
70
71
72
73  void
74
75  _quicksort (void *const pbase, size_t total_elems, size_t size,
76          __compar_d_fn_t cmp, void *arg)
77  {
78
79      /*details omitted*/
80        while (STACK_NOT_EMPTY)
81          {
82            char *left_ptr;
83            char *right_ptr;
84        /* Select median value from among LO, MID, and HI. Rearrange
85
86           LO and HI so the three values are sorted. This lowers the
87
88           probability of picking a pathological pivot value and
89
90           skips a comparison for both the LEFT_PTR and RIGHT_PTR in
91
92           the while loops. */
93        char *mid = lo + size * ((hi - lo) / size >> 1);
94        if ((*cmp) ((void *) mid, (void *) lo, arg) < 0)
95          SWAP (mid, lo, size);
96
97        if ((*cmp) ((void *) hi, (void *) mid, arg) < 0)
98          SWAP (mid, hi, size);
99
100       else
101         goto jump_over;
102
103       if ((*cmp) ((void *) mid, (void *) lo, arg) < 0)
104         SWAP (mid, lo, size);
105
106       jump_over:;
107       left_ptr  = lo + size;
108       right_ptr = hi - size;
109       /* Here's the famous collapse the walls'' section of quicksort.
110          Gotta like those tight inner loops!  They are the main reason
111          that this algorithm runs much faster than others. */
112       do
113         {
114           while ((*cmp) ((void *) left_ptr, (void *) mid, arg) < 0)
115             left_ptr += size;
```

```
116
117           while ((*cmp) ((void *) mid, (void *) right_ptr, arg) < 0)
118             right_ptr -= size;
119
120           if (left_ptr < right_ptr)
121             {
122               SWAP (left_ptr, right_ptr, size);
123               if (mid == left_ptr)
124                 mid = right_ptr;
125               else if (mid == right_ptr)
126                 mid = left_ptr;
127               left_ptr += size;
128               right_ptr -= size;
129             }
130
131           else if (left_ptr == right_ptr)
132             {
133               left_ptr += size;
134               right_ptr -= size;
135               break;
136             }
137         }
138
139       while (left_ptr <= right_ptr);
140           /* Set up pointers for next iteration.  First determine whether
141              left and right partitions are below the threshold size.
    If so,
142              ignore one or both.  Otherwise, push the larger partition's
143              bounds on the stack and continue sorting the smaller one. */
144           if ((size_t) (right_ptr - lo) <= max_thresh)
145             {
146               if ((size_t) (hi - left_ptr) <= max_thresh)
147         /* Ignore both small partitions. */
148                 POP (lo, hi);
149               else
150         /* Ignore small left partition. */
151                 lo = left_ptr;
152             }
153           else if ((size_t) (hi - left_ptr) <= max_thresh)
154
155         /* Ignore small right partition. */
156             hi = right_ptr;
157           else if ((right_ptr - lo) > (hi - left_ptr))
158             {
159           /* Push larger left partition indices. */
160               PUSH (lo, right_ptr);
161               lo = left_ptr;
162             }
163           else
164             {
```

```
165             /* Push larger right partition indices. */
166                 PUSH (left_ptr, hi);
167                 hi = right_ptr;
168               }
169           }
170       }
171
172        /*details omitted*/
173
174       if (tmp_ptr != base_ptr)
175         SWAP (tmp_ptr, base_ptr, size);
176       /* Insertion sort, running from left-hand-side up to right-hand-side.
    */
177
178       run_ptr = base_ptr + size;
179       while ((run_ptr += size) <= end_ptr)
180         {
181           tmp_ptr = run_ptr - size;
182           while ((*cmp) ((void *) run_ptr, (void *) tmp_ptr, arg) < 0)
183             tmp_ptr -= size;
184
185           tmp_ptr += size;
186           if (tmp_ptr != run_ptr)
187               {
188                   char *trav;
189                   trav = run_ptr + size;
190                   while (--trav >= run_ptr)
191                     {
192                       char c = *trav;
193                       char *hi, *lo;
194
195                       for (hi = lo = trav; (lo -= size) >= tmp_ptr; hi = lo)
196
197                         *hi = *lo;
198
199                       *hi = c;
200
201                     }
202               }
203         }
204     }
205 }
```

For the sake of convenience, we describe the basic algorithms used there, Merge sort, Insertion sort and Quick Sort.

2.2 Algorithm1 : Merge Sort

2.2.1 General

Algorithm 1 is Merge Sort. Merge Sort is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Merge Sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.
Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).

2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

2.2.2 Correctness

The correctness of merge sort is very simple to prove. Merge Sort uses a divide and conquer way. If $N = 1$, there is only one element to sort, and the correctness is self-evident. Otherwise, recursively mergesort the first and the second half. When two sorted arrays are here, compare the two top element and put the smaller one in the new array. It is guaranteed that in the new array the last one is smaller than the next one, that is, the new array is a sorted array. So the correctness is proved in a Mathematical Induction way.

2.2.3 Implementation in Pseudo Code

Code 2: Merge Sort in Pseudo Code

```
1  function MergeSort(a[], begin, end)
2      tempArray = array[1..end−begin+1];
3      left := begin;
4      right := end;
5      center := (left+right)/2;
6      tempP := 1;
7      if(left < right) then
8          MergeSort(a, left, center);
9          MergeSort(a, center+1, right);
10         Lp := left;
11         Rp := center+1;
12         while(Lp <= center && Rp <= right)
13           if a[Lp] <= a[Rp] then
14             tempArray[tempP++] = a[Lp++];
15           else
16             tempArray[tempP++] = a[Rp++];
17         while Lp <= center do
18             temp[tempP++] = a[Lp++];
19         while Rp <= right do
20             temp[tempP++] = a[rP++];
```

```
21          for 1:= 1 to end-begin+1 do
22              a[left] = temp[i];
23              left++;
```

2.2.4 Implementation in C

Code 3: Merge Sort in C

```
1  void MergeSort(int a[], int begin, int end){
2      int* tempArray = (int*)malloc(sizeof(int)*(begin-end+1));
3      int left = begin; int right = end;
4      int center = (left+right)/2;
5      int tempP = 0;
6      if(left < right){
7          MergeSort(a, left, center);
8          MergeSort(a, center+1, right);
9          int Lp = left; int Rp = center+1;
10         while(Lp < center&& Rp < right){
11             if(a[Lp] < a[Rp])  tempArray[tempP++] = a[Lp++];
12             else  tempArray[tempP++] = a[Rp++];
13         }
14         for(int i = 0; i < end-begin+1; i++){
15             a[left++] = tempArray[i];
16         }
17     }
18     free(tempArray);
19 }
```

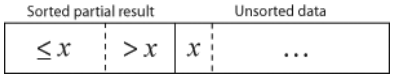
2.3 Algorithm2 : Insertion Sort

2.3.1 General

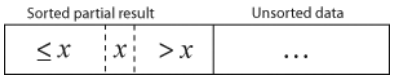
Insertion sort is a simple sorting algorithm that builds the final sorted array(or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

1. simple implemention
2. efficient for (quite) small data sets
3. More efficient in practice than most other simple quadratic
4. Adaptive, efficient for already substantially sorted data sets
5. stable
6. in-place only require a constant amount $O(1)$ of additional memory space
7. online, an sort a list as it receives it

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.
Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.
The resulting array after k iterations has the property where the first k + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:



becomes



with each element greater than x copied to the right as it is compared against x.

2.3.2 Correctness

Insertion sort actually divides the array into two parts, the sorted part and the unsorted part. Each time insertion sort found the smallest element and put it into the end of sorted part, which ensures that in Kth time the Kth smallest element finds its correct position. When $K = N$ the array is sorted.

2.3.3 Implementation in Pseudo Code

Code 4: Insertion Sort in Pseudo Code

```
1  function InsertionSort(a[])
2      for i = 1 to length(a)
3          x = a[i]
4          j = j−1
5          while j >= 0 and a[j] > x
6              a[j+1] = a[j]
7              j = j − 1
8          end while
9          a[j+1] = x
10     end for
```

2.3.4 Implementation in C

Code 5: Insertion Sort in C

```c
void InsertionSort(int a[], int n){
    for(int i = 0; i < n; i++){
        int x = a[i];
        int j = i-1;
        while(j>=0&&a[j]>x){
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = x
    }
}
```

2.4 Algorithm3 : Quick Sort

2.4.1 General

Algorithm2 is Quick Sort. Quick Sort is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1959. It is still a commonly used algorithm for sorting.
Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. In efficient implementations it is not a stable sort, meaning that the relative order of equal sort items is not preserved. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting.
Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.
The steps are:

1. Pick an element, called a pivot, from the array.
2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which never need to be sorted.
The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

2.4.2 Correctness

Quick sort is similar to mergesort, which is also a divide and conquer way. When $N = 1$, the correctness is self-evident. If $N! = 1$, the array is divided into two parts, and all elements in one part is larger than a certain element, all elements in another part is smaller than the certain elements. This going recursively finally makes the last element is always smaller than the next element. Because the adjacent elements will finally be divided into two different part, and the part near the back is always larger than the part near the top.

2.4.3 Implementation in Pseudo Code

Code 6: Quick Sort in Pseudo Code

```
1  function QuickSort(a[], left , right)
2      center := (left+right)/2;
3      if a[left] > a[center] then
4          swap a[left] with a[center]);
5      if a[left] > a[right] then
6          swap a[left] with a[right];
7      if a[center] > a[right] then
8          swap a[center] with a[right];
9      swap a[center] with a[right-1];
10     pivot = a[right-1];
11     i := left - 1;
12     j := right + 1;
13     loop forever
14         do
15             i:= i+1
16         while A[i] < pivot
17         do
18             j:=j-1
19         while a[j] > pivot
20         if i>= j then
21             break;
22         swap A[i] with A[j]
23     QuickSort(A, left, center);
24     QuickSort(A, center+1, right);
```

2.4.4 Implementation in C

Code 7: Quick Sort in C

```
1  void QuickSort(int a[], int left, int right){
2      if(left < right){
3          int center = (left+right)/2;
4          if(a[left] > a[center]){
5              int temp = a[left]; a[left] = a[center]; a[center]=temp;
6          }
```

```
7           if(a[left] > a[right]){
8               int temp = a[left]; a[left] = a[right]; a[right]=temp;
9           }
10          if(a[right] > a[center]){
11              int temp = a[right]; a[right] = a[center]; a[center]=temp;
12          }
13          int pivot = a[center];
14          int temp = a[center]; a[center]=a[right-1];a[right-1]=temp;
15          int Lp = left, Rp = right-1;
16          for(;;){
17              while(a[++Lp]>pivot){}
18              while(a[++Rp]<pivot){}
19              if(Lp<Rp){
20                  int temp = a[Lp]; a[Lp] = a[Rp]; a[Rp] = temp;
21              }else{
22                  break;
23              }
24          }
25          temp = a[i]; a[i]=a[right-1];a[right-1]=temp;
26          QuickSort(a, left, i-1);
27          QuickSort(a, i+1, right);
28      }
29  }
```

3 Testing Results

3.1 Test Cases

In this project, we used the following cases to test the correctness and the performance of our program:

3.1.1 Problem Sample

this test case is provided by the original problem, which is used for checking the correctness of the program in general cases.

```
16 7
JH007BD 18:00:01 in
ZD00001 11:30:08 out
DB8888A 13:00:00 out
ZA3Q625 23:59:50 out
ZA133CH 10:23:00 in
ZD00001 04:09:59 in
JH007BD 05:09:59 in
ZA3Q625 11:42:01 out
JH007BD 05:10:33 in
ZA3Q625 06:30:50 in
JH007BD 12:23:42 out
ZA3Q625 23:55:00 in
JH007BD 12:24:23 out
ZA133CH 17:11:22 out
JH007BD 18:07:01 out
DB8888A 06:30:50 in
05:10:00
06:30:50
11:00:00
12:23:42
14:00:00
18:00:00
23:59:00
```

3.1.2 Smallest Boundary Sample

This test case is generated manually, which is used for check the vulnerability and stablity of the problem when facing special situation general cases.

```
2 1
JH007BD 00:00:01 in
JH007BD 00:00:02 out
00:00:03
```

3.1.3 Out Records not Paired Sample

In this test case, except the guaranteed one, all of other records' status are *out* and no *in* records to pair with them. What's more, there are some cars have

out record before *in* record which should be ignored. We use this test case to test whether our program ignores any *out* records which are not paired with an *in* record.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
5798 16178
ECV0N74 12:39:03 in
ECV0N74 12:39:04 out
P9Q7S08 19:03:34 out
...
15:16:07
00:36:13
03:39:45
...
```

3.1.4 In Records not Paired Sample

In this test case, except the guaranteed one, all of other records' status are *in* and no *out* records to pair with them. We use this test case to test whether our program ignores any *in* records which are not paired with an *out* record.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
9970 830
IEK0W75 12:40:45 in
IEK0W75 12:40:46 out
87Q2R6H 13:53:28 in
...
12:34:01
20:20:07
14:53:16
...
```

3.1.5 Same Time Sample

In this test case, although every record doesn't have the same *in* or *out* time, they parked for the same time. For convenience, we set all the cars park for 1 second. We use this test case to test the output of plate numbers of longest-park-time cars.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
8203 31194
Q41Q992 19:24:24 in
2DB00C5 14:26:17 in
ANU03FC 02:04:36 in
...
14:26:36
19:29:52
```

```
12:40:27
...
```

3.1.6 Query as In Time Sample

In this test case, all the query time is the same as every *in* record. What's more, we let every car only parked for 1 second, which can let us see directly that the result of query search is correct.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
7568 28081
N80ALUL 19:55:53 in
R92X293 12:53:53 in
PB8T341 23:21:19 in
...
19:55:53
12:53:53
23:21:19
...
```

3.1.7 Query as Out Time Sample

In this test case, all the query time is the same as every *out* record. What's more, we let every car only parked for 1 second, which can let us see directly that the result of query search is correct.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
9054 3406
TU5WA93 05:54:10 in
HOI7PHP 04:52:10 in
7G9QI58 07:06:14 in
...
05:54:11
04:52:11
07:06:15
...
```

3.1.8 Total Random Sample

In this test case, we just generate data randomly but ensure that there are no illegal records.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
2406 11201
JC1K2U7 14:14:59 out
50II4S8 20:09:56 out
451CK6O 22:34:47 out
```

```
...
08:34:54
14:40:49
04:02:01
...
```

3.1.9 Largest Complex Sample

In this test case, we combine all of the situation above, we have many unpaired records and paired records. Besides, we have cars, some of which have the same park-time and the other have different park-time. What's more, the query is random.
This test case is too larger, we only provide part of the input in the report, you can find the full case in the same folder.

```
10000 80000
U333S2U 17:24:42 in
04R2775 10:30:51 out
GCHN763 21:38:02 out
...
21:42:05
22:09:18
02:39:42
...
```

3.2 Test Output

According our test, our program passed all test case correctly and quickly, here is the output of each test case generated by our program. However, some of the result is too larger, we only provide part of the output of the program, you can find the full case in the same folder.

3.2.1 Problem Sample

```
1
4
5
2
1
0
1
JH007BD ZD00001 07:20:09
```

3.2.2 Smallest Boundary Sample

```
0
JH007BD 00:00:01
```

3.2.3 Out Records not Paired Sample

```
0
0
0
...
ECV0N74 00:00:01
```

3.2.4 In Records not Paired Sample

```
0
0
0
...
IEK0W75 00:00:01
```

3.2.5 Same Time Sample

```
0
0
0
...
00105TT 004D75B 0055STT 0059KAJ 008IUKG 008K4LF ...
```

3.2.6 Query as In Time Sample

```
1
1
1
...
0015M97 002A92F 0048H6H 005N50A ...
```

There is some results are 2 for query. That's because at the same time, there is another car coming into campus.

3.2.7 Query as Out Time Sample

```
0
0
0
...
000N0B5 002PVS7 007EE12 007Q056 009K4V0 ...
```

There is some results are 1 for query. That's because at the same time, there is another car coming into campus.

3.2.8 Total Random Sample

```
182
182
108
77
...
L7PMKO7 22:22:51
```

3.2.9 Largest Complex Sample

```
170
149
191
255
...
2S02EGF 19:52:29
```

3.3 Running Time

After the validation of the correctness of our program, we also test the running time of our program, through the test script written in C, we got the running time of each case

```
Problem Sample : 0ms
Smallest Boundary Sample : 0ms
In Records not Paired Sample : 8 ms
Out Records not Paired Sample : 0 ms
Query as In Time Sample : 21 ms
Query as Out Time Sample : 11 ms
Same Time Sample : 24 ms
Total Random Sample : 7 ms
Largest Complex Sample : 63 ms
```

All of the result is tested on the following computer:
Hardware: Inter Core-i5 5200U / 8G DDR3 / SSD
System: Windows 8.1
Compiler: gcc version 6.2.1 20160830 (GCC)
Compile Command: g++ -std=c++11

4 Analysis and Comments

We first use theoretical analysis to deduce the growth of function of a certain algorithm, and then do validation in terms of the implementation in C code. Finally, we make comparison between the result from analysis and the result from performance test, in order to make it be accurate.

4.1 Quick Sort

4.1.1 Time Complexity

Like mergesort, quicksort is recursive, and hence, its analysis requires solving a recurrence formula. We will do the analysis for a quicksort, assuming a random pivot and no cutoff for small files. We will take $T(0) = T(1) = 1$. The running time of quicksort is equal to the running time of the two recursive calls plus the linear time spent in the partition. This gives the basic quicksort relation

$$T(N) = T(i) + T(N - i - 1) + cN \tag{1}$$

where $i = |S_1|$(the divided part after pivot chosen) is the number of elements in S_1.

Worst-Case Analysis

The pivot is the smallest element, all the time. Then $i = 0$ and if we ignore $T(0) = 1$, which is insignificant, the recurrence is

$$T(N) = T(N - 1) + cN, N > 1 \tag{2}$$

We telescope, using Equation (1) repeatedly. Thus

$$T(N - 1) = T(N - 2) + c(N - 1) \tag{3}$$

$$T(N - 2) = T(N - 3) + c(N - 2) \tag{4}$$

$$\vdots$$

$$T(2) = T(1) + c(2) \tag{5}$$

Adding up all these equations yields

$$T(N) = T(1) + c\sum_{i=2}^{N} i = O(N^2) \tag{6}$$

Best-Case Analysis

In the best case,the pivot is in the middle. To simplify the math, we assume that the two subarrays are each exactly half the size of the original, and although this gives a slight overestimate, this is acceptable because we are only interested in a Big-Oh answer.

$$T(N) = 2T(N/2) + cN \tag{7}$$

Divided both sides of Equation (7) by N.

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c \tag{8}$$

We will telescope using this equation.

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + c \tag{9}$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + c \tag{10}$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1}{1} + c \tag{11}$$

We add all the equation from (7) to (11) and note that there are log$N$ of them:

$$\frac{T(N)}{N} = \frac{T(1)}{1} + c\log N \tag{12}$$

which yields

$$T(N) = cN\log N + N = O(N\log N) \tag{13}$$

Average-Case Analysis

This is the most difficult part. For the average case, we assume that each of the file sizes for S_1 is equally likely, and hence has probabiliyu $1/N$. This assumption is actually valid for our pivoting and partitioning strategy, but it is not valid for some others. Paetitioning strategies that do not preserve the randomness of the subfiles cannot use this analysis. Interestingly, these strategies seem to result in programs that take longer to run in practice.
With this assumption, the average value of $T(i) =$, and hence $T(N-i-1)$, is $(1/N)\sum_{j=0}^{N-1} T(j)$. Equation (1) then becomes

$$T(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} T(j)\right] + cN \tag{14}$$

If Equation (14) is multiplied by N, it becomes

$$NT(N) = 2\left[\sum_{j=0}^{N-1} T(j)\right] + cN^2 \tag{15}$$

We need to remove the summation sign to simplify matters. We note that we can telescope with one more equation.

$$(N-1)T(N-1) = 2\left[\sum_{j=0}^{N-2} T(j)\right] + c(N-1)^2 \tag{16}$$

If we subtract (16) from (15), we obtain

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c \tag{17}$$

We rearrange terms and drop the insignificant $-c$ on the right, obtaining

$$NT(N) = (N+1)T(N-1) + 2cN \tag{18}$$

We now have a formula for $T(N)$ in terms of $T(N-1)$ only. Again the idea is to telescope, but Equation (18) is in the wrong form. Divide Equation (18) by N(N+1):

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1} \tag{19}$$

Now we can telescope.

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N} \tag{20}$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N-1} \tag{21}$$

$$\vdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3} \tag{22}$$

Adding Equations (19) through (22) yields

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2\sum_{i=3}^{N+1} \frac{1}{i} \tag{23}$$

The sum is about $\log_e(N+1) + \gamma - \frac{3}{2}$, where $\gamma \approx 0.577$ is known as Euler's constant, so

$$\frac{T(N)}{N+1} = O(\log N) \tag{24}$$

And so

$$T(N) = O(N \log N) \tag{25}$$

Although this analysis seems complicated, it really is not – the steps are natural once you have seen some recurrence relations.

4.1.2 Space Complexity

The space used by quicksort depends on the version used. The in-place version of quicksort has a space complexity of $O(\log n)$, even in the worst case, when it is carefully implemented using the following strategies:

- in-place partitioning is used. This unstable partition requires $O(1)$ space.
- After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most$O(\log n)$ space. Then the other partition is sorted using tail recursion or iteration, which doesn't add to the call stack. This idea, as discussed above, was described by R. Sedgewick, and keeps the stack depth bounded by $O(\log n)$.

Quicksort with in-place and unstable partitioning uses only constant additional space before making any recursive call. Quicksort must store a constant amount of information for each nested recursive call. Since the best case makes at most $O(\log n)$ nested recursive calls, it uses $O(\log n)$ space. However, without Sedgewick's trick to limit the recursive calls, in the worst case quicksort could make $O(n)$ nested recursive calls and need $O(n)$ auxiliary space.

Another, less common, not-in-place, version of quicksort uses $O(n)$ space for working storage and can implement a stable sort. The working storage allows the input array to be easily partitioned in a stable manner and then copied back to the input array for successive recursive calls. Sedgewick's optimization is still appropriate.

4.2 Other parts of program

Besides quicksort, in our program, we use some other procedure to get the final answer. In order to find the records that is valid and meanwhile count the total stop time and update the max time, we use a loop whose time complexity is $O(N)$. Besides, in order to read query and count the number of cars, we use another loop whose time complexity is $O(N+K)$. In terms of space complexity, most of space is to store the records and queries which is $O(N+K)$.

4.3 Validation

In order to verify the Conclusion from the theoretical analysis of time complexity, we using some test script to test our program, in terms of the variation of the input data size, we can get a N-T relation graph, by the analysis the figure, we can verify the Conclusion from the theoretical deduction.

4.4 Validation

In order to verify the Conclusion from the theoretical analysis of time complexity, we using some test script to test our program, in terms of the variation of the input data size, we can get a N-T relation graph, by the analysis the figure, we can verify the Conclusion from the theoretical deduction.

4.4.1 Validation Result

Because quicksort runs so quickly that it takes less than a tick to finish, we repeat it calls for 1000 times to obtain a total run time to get more accurate data. By running the test script, we generate input data randomly with different number of records, in order to test the time complexity of major function – quciksort. After running the test script, we can get the following result.

N	100	500	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Time (ms)	24	236	576	1172	1832	2732	3607	4830	5072	6308	6807	8363
	27	292	580	1172	1877	2916	3731	4291	5147	6539	6891	8010
	27	244	553	1372	1841	2741	3540	4306	5359	6419	7615	7856
Average	26	257	570	1239	1850	2796	3626	4476	5193	6422	7104	8076

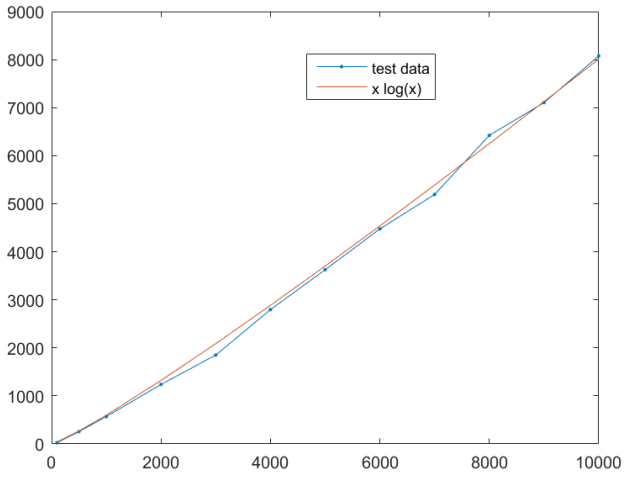


Figure 1: Performance on Random Data

we can found that, the result of our $N - T$ function is similar to $n\log n$ function, which can prove that our program nearly has a time complexity of $O(N\log N)$, and from the theoretical analysis, we can found that the time complexity of quciksort is $O(N\log N)$, which is equal, which is satisify the result of our theoretical analysis.

4.5 Comments

In general, the test result quite matches our expectation. Our Program perfectly finished the requirement of the given problem, and work out the result correctly and efficient, according to the theoretical analysis, we can prove the time Complexity and the space consumption of out program is acceptable, which is required by the online judge system, and the result of the validation shows that the result of our theoretical analysis is correct at a large probability.

5 Appendix

5.1 Source Code

See this subsection on the additional ‘project3.c‘ file.

```
1  // Project 3 : Cars on Campus
2  // This Program is written in the ISO C++ 11 standard
3  // (ISO/IEC 14882:2011, see on
4  //  http://www.iso.org/iso/catalogue_detail.htm?csnumber=50372)
5  // We strongly recommend you using the latest compiler to
6  // compile this program, for instance, you can use
7  // > g++ 4.7 or later, and using compile option "-std=c++11"
8  // > clang 3.1 or later, and using compile option "-std=c++11"
9  // > visual studio 2013 or later
10
11
12 // The following headers are just C++ style standard C
13 // Library, which is not contains STL (Standard Template
14 // Library) files, which is forbidden by our course
15 #include <cstdio>
16 #include <cstring>
17 #include <cstdlib>
18
19
20 // The array size, which is determined in compiler and
21 // will be static allocated during the compile time,
22 // for reducing the time cost for dynamic memory allocation
23 const int N_MAX = 20000 + 100; // The max value of N ( records )
24 const int K_MAX = 80000 + 100; // The max value of K ( querys )
25
26 // The Record Entity Mode;
27 // Each Record instance stands for an unique in/out record;
28 struct Record {
29
30     enum class Direction{ // The enumerate class stands for the direction prope
31         in, out           // of an certain record
32     } direction;
33
34     bool useful;          // The flag indicated whether this record is vaild, w
35                           // is decided in the filter percedure
36     int time_stamp;       // Record Timestamp, a signed integer in [0, 86400)
37
38     char * plate_str;     // The plate string in C style, which is fast and lig
39     Record(const char * plate_str, Direction direction, int time_stamp)
40             : direction(direction), time_stamp(time_stamp), useful(false) {
41
42         this->plate_str = new char[strlen(plate_str)]; // allocate new memory s
43         strcpy(this->plate_str, plate_str);            // copy the original str
44
45     }
```

```
46
47      ~Record(){
48          delete[] plate_str; // release the memory space
49      }
50  };
51
52  // The array of all records
53  Record * records[N_MAX];
54
55  // A simple vector implementation for storage the
56  // plate str of which has the max length stop time
57  const char * max_length_str[K_MAX];         // The space of vector
58  int max_length_str_size = 0;                // The current size of vector
59
60  // The max length of stop time
61  int max_length = -1;
62
63  // The plate string of plate under processing currently
64  const char * current_plate_str = nullptr;
65  // The count of stop time of plate under processing currently
66  int current_plate_length = 0;
67
68  // Comparing the stop time of current plate with the
69  // max stop length of plate currently, and update the
70  // value of "max_length" and corresponding "max_length_str"
71  // vector
72  void deal_max_length(){
73      if( current_plate_length > max_length ){        // replace the old length
74          max_length = current_plate_length;
75          max_length_str_size = 0;
76          max_length_str[++max_length_str_size] = current_plate_str;
77      }else if( current_plate_length == max_length ){ // append the current strin
78          max_length_str[++max_length_str_size] = current_plate_str;
79      }
80  }
81
82  // Read the time string format of hh::mm::ss from the
83  // standard input, and return a value in well-defined Timestamp,
84  // which a signed integer in [0, 86400)
85  int read_time_stamp() {
86      int hh, mm, ss;
87      scanf("%d:%d:%d", &hh, &mm, &ss);
88      return hh * 3600 + mm * 60 + ss;
89  }
90
91  int main() {
92      char buffer[16], direction[16]; //The buffer for reading string
93
94      int n = 0, k = 0; //The n ( the number of records ) and k (the number of qu
95      scanf("%d%d", &n, &k);
```

```
96
97      //Read all record from the standard input
98      for (int i = 1; i <= n; i++) {
99          scanf("%s", buffer);
100         int time_stamp = read_time_stamp();
101         scanf("%s", direction);
102         if (direction[0] == 'i') // Detected an in record
103             records[i] = new Record(buffer, Record::Direction::in, time_stamp);
104         else                     // Detected an out record
105             records[i] = new Record(buffer, Record::Direction::out, time_stamp)
106     }
107
108     // Sort all of the records according to the plate_str
109     // first, which is in order to integrate all of the
110     // records of same plate together, then sort the
111     // the records of the same plate by time ascending
112     // order
113     qsort(&records[1], n, sizeof(Record *),
114           [](const void * a, const void * b) ->int{
115             Record & aa = **(Record**)a;
116             Record & bb = **(Record**)b;
117             int cmp_ret = strcmp(aa.plate_str, bb.plate_str);
118             if(  cmp_ret == 0 )
119                 return aa.time_stamp > bb.time_stamp ? 1: -1;
120             else
121                 return cmp_ret > 0 ? 1 : -1;
122     });
123
124     // Finding the records that is vaild, which has
125     // corresponding in/out pair sequentially, at the
126     // mean time, count the total stop time for a centain
127     // plate, and update the max_length of all of the
128     // records
129     for (int i = 1; i < n ; i++) {
130         auto &record_in = *records[i];
131         auto &record_out = *records[i + 1];
132         // find record pairs satistify the following condition
133         if( record_in.direction == Record::Direction::in
134             && record_out.direction == Record::Direction::out
135             && strcmp(record_in.plate_str, record_out.plate_str) == 0){
136             if(!(current_plate_str && strcmp(record_in.plate_str, current_plate
137                 deal_max_length(); // update the max_length
138                 current_plate_str = record_in.plate_str;
139                 current_plate_length = 0;
140             }
141             current_plate_length += record_out.time_stamp - record_in.time_stam
142             record_in.useful = true;   // mark as vaild
143             record_out.useful = true;  // mark as vaild
144         }
145     }
```

```
146
147     deal_max_length(); // update the max_length
148
149     // Rearrange all of the records in the ascending
150     // order of time, which is useful to get a linear
151     // time cost to calculate the total number of car
152     // of a certain time sequentially
153     qsort(&records[1], n, sizeof(Record *),
154           [](const void * a, const void * b) ->int {
155               Record & aa = **(Record**)a;
156               Record & bb = **(Record**)b;
157               return aa.time_stamp > bb.time_stamp ? 1: -1;
158           });
159
160     // the time stamp of each query from standard input
161     int query_time_stamp = 0;
162
163     // the states of the current time that are processing
164     // linear seqentially, which contains the current time
165     // and the the total number of car currently.
166     int now_time_stamp = 1, now_car_count = 0;
167
168
169     // Reading the timestamp of each query from the standard input
170     // And linear increaing the time to calcate the total of car
171     // at a ascending time order
172     for(int i = 1 ; i <= k; i++){
173         query_time_stamp = read_time_stamp();
174         //process all of the record before the current time
175         while( now_time_stamp <= n
176             && records[now_time_stamp]->time_stamp <= query_time_stamp){
177             if(records[now_time_stamp]->useful){
178                 // change the count of car currently
179                 if(records[now_time_stamp]->direction == Record::Direction::in)
180                     now_car_count++;
181                 }else{
182                     now_car_count--;
183                 }
184             }
185             now_time_stamp++;
186         }
187         printf("%d\n", now_car_count);
188     }
189
190     // Ouput all of the string of plates in all records which
191     // has the same and the greatest stop time ascendingly
192     for (int i = 1 ; i <= max_length_str_size; i++) {
193         printf("%s ", max_length_str[i]);
194     }
195
```

```
196     // Output the max stop time of all plate
197     printf("%02d:%02d:%02d\n", max_length / 3600, (max_length % 3600) / 60, max
198
199     // return SUCCESSFUL singal to the operating system
200     return 0;
201 }
```

5.2 Declaration

We hereby declare that all the work done in this project titled "Cars on Campus" is of our independent effort as a group.

5.3 Duty Assignments

Programmer: Programmer
Tester: Tester
Report Writer: Writer