**Hard Arrangement of Computation Tasks**

**朱理真**

3190101094

**Date: 2021-01-02**

Index

[Chapter 1: Introduction 2](#_Toc59962263)

[Chapter 2: Algorithm Specification 2](#_Toc59962264)

[Chapter 3: Testing Results 6](#_Toc59962265)

[Chapter 4: Analysis and Comments 23](#_Toc59962266)

[Appendix: Source Code (in C) 26](#_Toc59962267)

### Chapter 1: Introduction

Given the problem that there’s a lot of servers running at different time, and with all these servers’ records provided, we are asked to tell the longest time a task can run which cannot be stopped or interrupted. Also, given a bench of queries, how much starting times available for each computer task should be told.

The following is a formal description:

**Input:**

In the first line, there’s two integers, denoted n and k respectively, with n being the number of server records, and k being the number of queries. The following n lines each gives a server’s name and the open/close time.

The next k lines, each gives a time the task need to be run.

**Output:**

In the first line, give the longest time a task can be run in seconds. Then giving

each task’s total number of valid time points for starting the given computation task.

In my opinion, though the description of this issue seems confusing, it is not difficult, but need several sorts, to make the data in order. As for me, it is helpful to improve my familiarity of sorting algorithms.

### Chapter 2: Algorithm Specification

Data Structure

 record:

 \* records the start or end time of a server

1. **typedef** **struct** record{
2. **char** server[8];
3. **int** time;
4. record;

interval:

 \* records the start time and end time of a server.

 \* we don't save the name of the server.

1. **typedef** **struct** interval{
2. **int** begin, end;
3. }\*interval;

 times:

 \* times\_list:

  records all the continuous running times available for tasks.

 \* times\_sum\_list: sum the element in times\_list

 \* len: the length of list

1. **typedef** **struct** times{
2. **int**\* times\_list;
3. **int**\* times\_sum\_list;
4. **int** len;
5. }\*times;

Main Algorithm

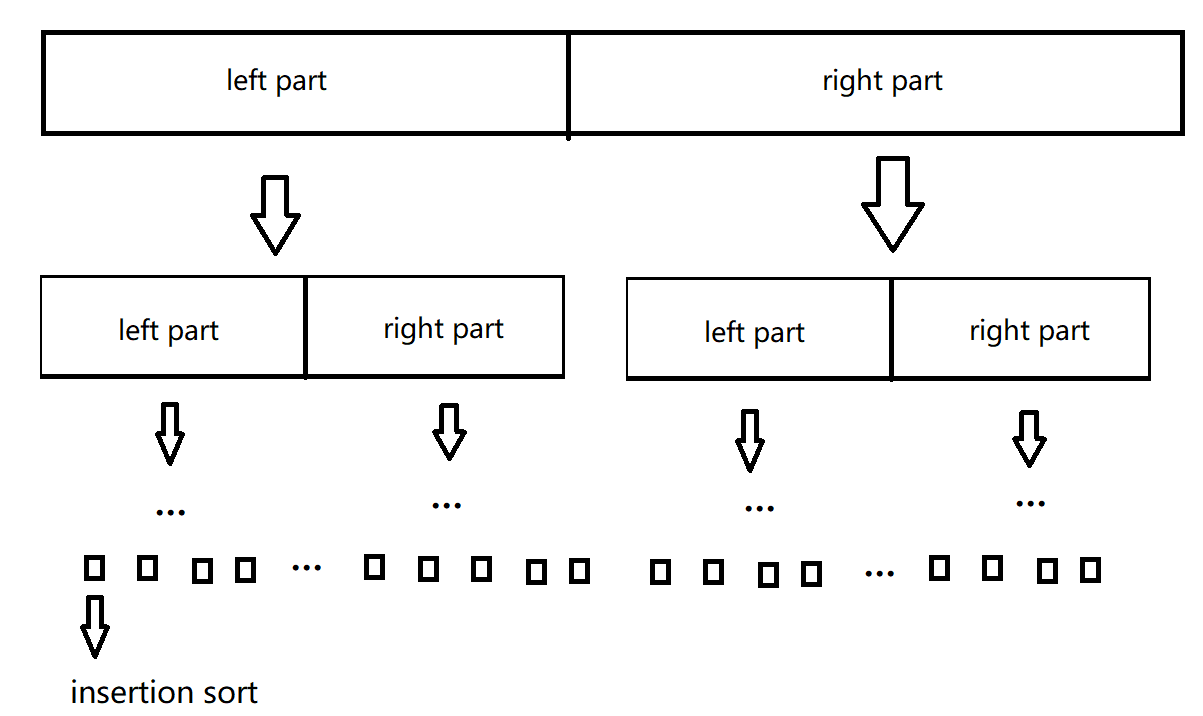
**main sturcture**:

1. Read the server's records from input
3. Create structure intervals from records
5. Merge intervals into structure times
7. process each query
9. give output

Key Algorithm

**Quick Sort**

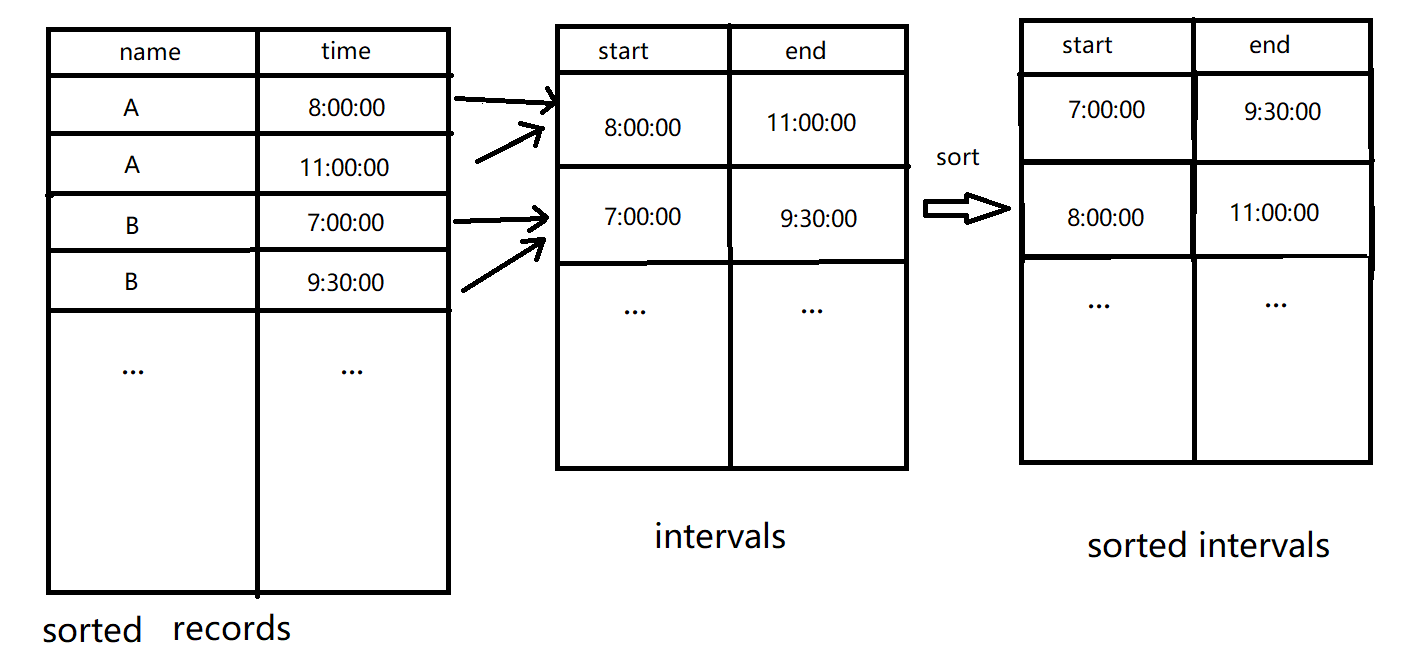
1. Quick Sort(array, left, right):
2. If right - left is too **short**
3. Use insertion sort instead
5. Compare the left, middle and the right element of the array,
6. Find the second largest one,
7. And set it as the element dividing the array into left and right part
9. Divide the array into left and right part
11. sort the left part
12. sort the right part



quick sort

**Create Intervals**

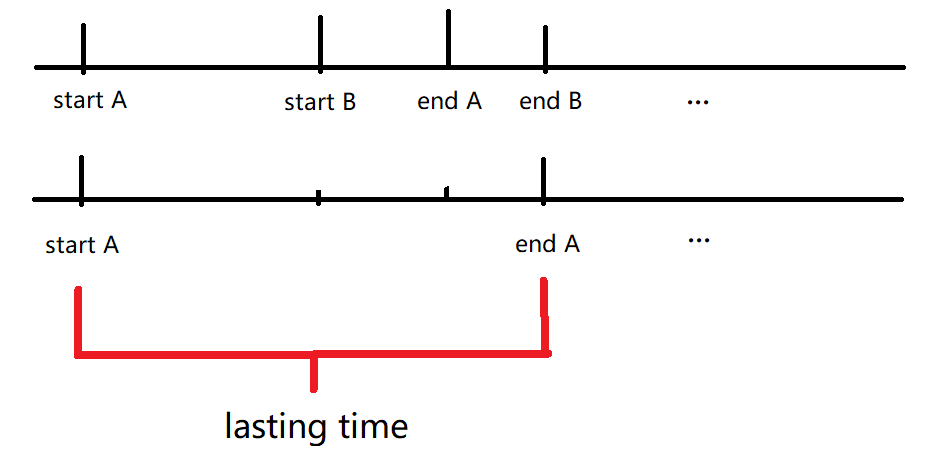
1. Create Intervals(records):
2. sort records first by their names, then(**if** equal) by their time.
3. **for** each two records with same server’s name
4. create an interval
5. let the earlier time of two records be the starting time
6. of the interval, later one be the ending time
8. sort the intervals by their starting times
9. **return** intervals;



Create Intervals

**Merge Intervals**

1. Merge Intervals(intervals):
3. **for** each two adjacent intervals
4. **if** they overlap, merge them together
5. repeat **this**, until no intervals overlap
7. list the lasting times of intervals, call them “time”s
9. sort times



merge intervals & lasting time

**Process Queries**

1. Process Queries(times, queries):
2. output the longest lasting time.
3. **for** each query, get their task’s running time
4. **for** each lasting time larger than the running time
5. start time available = lasting time -running time+1
6. sums all start time available and output it.

### Chapter 3: Testing Results

All inputs are also stored in txt file in the folder “document”. You should copy the data there to test the program. Inputs here just for read.

|  |  |
| --- | --- |
| TestCase1 | case purpose |
| Sample |
| expected result |
| 31801  1201  0  13202  20063  64597  13385  64373 |
| actual behavior |
| 31801  1201  0  13202  20063  64597  13385  64373 |
| possible cause |
|  |
| current status |
| pass |
| Input |
| 12 7  jh007bd 18:00:01  zd00001 11:30:08  db8888a 13:00:00  za3q625 23:59:50  za133ch 13:00:00  zd00001 04:09:59  za3q625 11:42:01  za3q625 06:30:50  za3q625 23:55:00  za133ch 17:11:22  jh007bd 23:07:01  db8888a 11:35:50  08:30:01  12:23:42  05:10:00  04:11:21  00:04:10  05:06:59  00:05:11 |
| TestCase2 | case purpose |
| Small size |
| expected result |
| 31898  28751  13457  28990  25007  5251  18659  31033  9124 |
| actual behavior |
| 31898  28751  13457  28990  25007  5251  18659  31033  9124 |
| possible cause |
|  |
| current status |
| pass |
| Input |
| 16 8  kuvemhb 05:15:25  rsgf4fp 07:34:30  foelzbl 07:04:37  g4l8xhf 05:41:36  kuvemhb 08:51:58  7es9x07 00:09:16  rsgf4fp 03:24:11  lxvu3fa 08:59:15  7es9x07 08:05:18  yol0aop 04:15:44  q1fiq38 08:25:30  g4l8xhf 09:00:54  yol0aop 02:35:04  foelzbl 00:55:02  q1fiq38 08:34:28  lxvu3fa 02:10:17  00:52:28  05:07:22  00:48:29  01:54:52  07:24:08  03:40:40  00:14:26  06:19:35 |
| TestCase3 | case purpose |
| No intervals overlap |
| expected result |
| 21063  0  21738  0  37548 |
| actual behavior |
| 21063  0  21738  0  37548 |
| possible cause |
|  |
| current status |
| pass |
| Input |
| 10 4  aaabbbc 1:00:00  aaabbbc 1:30:00  faddfafff 2:32:00  faddfafff 4:45:59  2o4u0jf 6:00:00  2o4u0jf 4:46:00  09aufh9 7:48:03  09aufh9 10:40:20  32jfad0f 13:32:09  32jfad0f 19:23:12  6:06:07  1:20:32  12:32:33  00:00:24 |
| TestCase4 | case purpose |
| All intervals can be merged to run a task |
| expected result |
| 19200  0  10231  16783 |
| actual behavior |
| 19200  0  10231  16783 |
| possible cause |
|  |
| current status |
| pass |
| Input |
| 6 3  fa0909f 6:20:00  pqepi13 7:00:00  cvzmwi2 8:20:37  pqepi13 8:30:00  cvzmwi2 10:20:37  fa0909f 11:40:00  7:32:32  2:29:30  0:40:18 |
| TestCase5 | case purpose |
| All the intervals have same start time and end time |
| expected result |
| 5854  0  1045  5121 |
| actual behavior |
| 5854  0  1045  5121 |
| possible cause |
|  |
| current status |
| pass |
| Input |
| 6 3  bdusj1l 04:48:39  2w66xo4 04:48:39  2w66xo4 06:26:13  i0fsin8 04:48:39  i0fsin8 06:26:13  bdusj1l 06:26:13  06:38:04  01:20:10  00:12:14 |
| TestCase6 | case purpose |
| Smallest size |
| expected result |
| 5543  0 |
| actual behavior |
| 5543  0 |
| possible cause |
|  |
| current status |
| pass |
| Input |
| 2 1  rlm5qlw 01:04:28  rlm5qlw 02:36:51  02:56:05 |
| TestCase7 | case purpose |
| Small size |
| expected result |
| 46  24707  23655  0  18036  15354  10986  11878  13982  6976  5519  3535  4114  9872  12374  31769  11038  15991  23878  2073  31852  24928  19407  25458  28486  3296  1250 |
| actual behavior |
| The same as expected |
| possible cause |
|  |
| current status |
| pass |
| Input |
| 60 40  t5pmtoy 04:51:46  tjt7wh3 01:44:05  rhjeqnv 08:31:43  o34c9fi 00:16:47  ioqu3wk 07:44:15  l97xfzo 00:30:23  xh9hgw8 00:52:00  4b4gia4 02:00:11  e0htdk4 07:46:16  kk2l20g 04:21:29  u3eg7ph 06:19:49  gxd6jen 02:32:04  gxd6jen 05:47:59  rhjeqnv 02:45:39  30qzkcj 00:56:22  n8zjumi 02:45:54  cet01zr 04:07:05  l97xfzo 06:33:04  g3ncnh0 06:50:56  3co6ia3 09:04:55  ivdqkwz 01:14:58  qn4ara6 01:58:55  ioqu3wk 00:54:07  n8zjumi 00:34:54  vk7x26i 02:39:22  g3ncnh0 00:01:26  dkhlmwx 04:09:14  ap9n1ko 04:45:10  qn4ara6 00:07:57  kyo2wdp 05:04:02  61ovohr 02:56:06  th839cy 08:19:13  cet01zr 01:54:36  vz6b3gh 04:34:52  61ovohr 06:41:46  ks9jxxz 08:47:00  0h5pv5j 08:47:41  kyo2wdp 00:26:41  o34c9fi 01:42:59  4b4gia4 00:44:42  th839cy 02:11:32  xh9hgw8 06:23:50  e0htdk4 02:39:42  rdzhzs1 07:50:20  ks9jxxz 07:32:10  tjt7wh3 06:04:16  5j4valq 08:55:15  ivdqkwz 03:17:43  5j4valq 07:05:12  dkhlmwx 02:16:42  3co6ia3 02:43:05  vk7x26i 07:23:13  kk2l20g 00:46:37  30qzkcj 05:06:53  0h5pv5j 01:43:30  t5pmtoy 02:56:28  u3eg7ph 01:25:36  ap9n1ko 05:00:38  rdzhzs1 03:01:46  vz6b3gh 03:13:36  05:48:31  06:13:09  01:57:05  06:17:39  03:05:12  07:10:22  08:01:03  08:33:03  02:35:48  05:19:15  04:17:02  05:15:21  01:01:10  09:02:44  02:11:43  02:29:15  09:03:56  04:02:54  04:47:36  06:00:24  05:45:32  05:10:28  07:07:14  07:31:31  08:04:35  07:54:56  06:18:58  05:37:16  00:14:01  05:59:32  04:36:59  02:25:32  08:28:57  00:12:38  02:08:02  03:40:03  01:59:12  01:08:44  08:08:34  08:42:40 |
| TestCase8 | case purpose |
| Smallest size |
| expected result |
| 30601  13978  28404  26965  21053  2163  26581  1333  30187  6109  11249  16266  6546  20  5515  17854  16562  21829  20235  9158  26047  3179  2927  19164  22141  6125  8305  29434  0  13560  3708  13080  4891  9628  16974  27686  14629  21416  7277  25287  12558  6213  2634  9169  5319  9299  15670  3975  9976  17798  19207  10131  22802  1087  17545  0  29973  22059  3494  20877  5921  24989  852  25145  25277  3499  15170  19437  8169  23305  26944  4454  11237  16808  13937  17518  27070  3852  5682  14108  7836 |
| actual behavior |
| The same as expected |
| possible cause |
|  |
| current status |
| pass |
| Input |
| 100 80  0knb2eh 05:46:51  4ssl66c 00:04:12  ps3ojwq 01:22:18  yg0rpxj 02:22:25  rw45zbn 02:20:10  4rp2l3g 01:51:51  w39eqce 06:46:07  rw45zbn 05:28:38  yg0rpxj 06:34:12  ps3ojwq 05:50:01  mv9i2v3 05:35:54  lxiyqpa 02:25:02  9evceb4 06:26:46  lp2xism 02:36:21  v292d6d 01:50:59  vpj72q2 07:35:26  ct5q8px 08:25:34  v5lya8l 00:48:45  u5hbbft 01:13:43  rd4drzt 01:09:52  q8rfjn5 01:35:29  a32xtod 03:52:25  lp2xism 00:21:51  2ihgmlv 07:35:42  0knb2eh 03:49:19  rcy1onw 06:37:40  vpj72q2 02:10:09  9evceb4 01:51:30  sp0s1n1 04:47:41  4aru76s 01:26:25  idm88dt 05:46:12  uh9wae5 01:33:30  lkh7dvd 06:27:02  u5hbbft 03:34:38  o6in3ja 00:15:21  ga6xr8p 04:49:10  2rjzk3k 06:59:01  ktytopr 05:43:20  uh9wae5 05:07:03  cwknayu 00:24:02  wre260y 03:32:28  rd4drzt 07:24:21  9bdljgm 05:12:31  p3umvtz 02:41:15  4aru76s 02:29:59  sp0s1n1 03:13:59  ptosbck 02:15:56  3mplz51 03:38:37  r5q5x1e 03:25:27  6gxykf1 01:07:58  ct5q8px 03:05:12  rcy1onw 02:53:13  grepelc 05:28:27  a32xtod 02:23:43  cwknayu 03:32:47  wre260y 04:10:25  q1grzu0 02:18:23  ktytopr 06:23:05  f5tvoob 02:04:13  ga6xr8p 01:22:07  r5q5x1e 06:31:45  o6in3ja 04:09:56  o02loh4 08:34:31  52g6p7r 00:20:43  kmoia20 05:27:24  o02loh4 01:20:14  p3umvtz 02:32:59  ncwdhhm 05:24:47  4rp2l3g 07:18:55  gl83jpe 04:47:15  lkh7dvd 00:37:25  2rjzk3k 07:11:18  9bdljgm 00:49:53  ncwdhhm 01:26:06  grepelc 04:11:41  kmoia20 03:29:24  ptosbck 06:19:48  679w06p 03:35:09  w39eqce 07:58:53  05sciz5 06:08:37  mv9i2v3 03:44:50  2ihgmlv 00:43:58  52g6p7r 03:22:18  kdj32w0 00:37:32  05sciz5 02:53:36  4s25rff 00:53:50  v292d6d 01:59:26  4ssl66c 00:12:02  lxiyqpa 00:33:15  f5tvoob 08:45:22  4s25rff 03:10:32  679w06p 05:32:12  q1grzu0 07:18:07  idm88dt 04:09:26  gl83jpe 07:34:23  v5lya8l 01:45:43  3mplz51 00:26:54  kdj32w0 05:30:37  6gxykf1 05:59:53  q8rfjn5 04:39:55  04:37:04  00:36:38  01:00:37  02:39:09  07:53:59  01:07:01  08:07:49  00:07:23  06:48:13  05:22:33  03:58:56  06:40:56  08:29:42  06:58:07  03:32:28  03:54:00  02:26:13  02:52:47  05:57:24  01:15:55  07:37:03  07:41:15  03:10:38  02:21:01  06:47:57  06:11:37  00:19:28  08:32:28  04:44:02  07:28:14  04:52:02  07:08:31  05:49:34  03:47:08  00:48:36  04:26:13  02:33:06  06:28:45  01:28:35  05:00:44  06:46:29  07:46:08  05:57:13  07:01:23  05:55:03  04:08:52  07:23:47  05:43:46  03:33:24  03:09:55  05:41:11  02:10:00  08:11:55  03:37:37  08:48:05  00:10:29  02:22:23  07:31:48  02:42:05  06:51:21  01:33:33  08:15:50  01:30:57  01:28:45  07:31:43  04:17:12  03:06:05  06:13:53  02:01:37  01:00:58  07:15:48  05:22:45  03:49:54  04:37:45  03:38:04  00:58:52  07:25:50  06:55:20  04:34:54  06:19:26 |
| TestCase9 | case purpose |
| Large size |
| expected result |
| Too long to show. See it in 9.out |
| actual behavior |
| The same as expected |
| possible cause |
|  |
| current status |
| pass |
| Input |
| Too long to show. See it in 9.in |
| TestCase10 | case purpose |
| Largest size |
| expected result |
| Too long to show. See it in 10.out |
| actual behavior |
| The same as expected |
| possible cause |
|  |
| current status |
| pass |
| Input |
| Too long to show. See it in 10.in |

### Chapter 4: Analysis and Comments

**Using the detailed pseudo code to analyze.**

Time Complexity:

**Analyze:**

**Input:**

**n:** the number of server records

**k:** the number of queries

**In function main:**

Normal sequential execution statements take O(1) time.

Function

read\_record,

CreateIntervals,

DestroryRecords,

MergeIntervals,

process\_query,

cat and

DestroyIntervals are called.

Therefore, T(main) =

T(read\_record) + T(CreateIntervals) + T(DestroryRecords) + T(MergeIntervals) + T(process\_query) + T(cat) + T(DestroyIntervals)

**The following is the time complexity analysis of the functions mentioned above.**

**In function read\_record:**

**Since there’s n records, O(n) time is taken.**

Therefore, T(read\_record) = O(n)

**In function** CreateIntervals**:**

**Since there’s for-loop(n/2) and two quick-sorts, T(**CreateIntervals)= O(n/2) + O(n/2\*log n) + O(n\*log n)=O(n\*log n)

**In function** DestroryRecords**:**

**Since there’s n records, O(n) time is taken.**

Therefore, T(DestroryRecords) = O(n)

**In function** MergeIntervals**:**

**Since there’s a for-loop(n/2) and a quick-sort,**

T(MergeIntervals) = O(n/2) + O(t\*log t) =O(n/2)+O(n\*log n)=O(n\*log n)

t: After Merging, the number of intervals.

**In function** process\_query**:**

**Since there’s a for-loop(k) and k binary-search(O(n)),**

T(process\_query) = O(k) + O(k\*log n) =O(k\*log n)

**In function** cat**:**

**Since there’s k outputs, T(cat)=O(k)**

Like DestroryRecords, T(DestroyIntervals)=O(n)

**Above all,**

**T=T(main) = O(k) + O(n) + O(n\*log n) + O(k\*log n) =**

**(n:** the number of server records, **k:** the number of queries)

**Space Complexity:**

**Analyze:**

**In the whole program:**

Array of intervals created, taking O(n/2) space

Array of records created, taking O(n) space

Structure times created, taking O(n)

Other local variables take O(1) space

However, there’s three quick-sorts with recursions.

In each quick-sort, we have

where

As for k, in the best case,

therefore,

and in the worst case,

Hence,

and,

Above all, S = =

**Possible Improvements:**

1)

make the names of structs more understandable.

I think the structure ‘interval’ and ‘times’ may make readers confusing. A solution is to describe the structure more precisely, though that would make the name be longer.

2)

Using hash table to save the records. With hash table, the sort of records is not needed because when we travel the list of records, we can quickly find the respond start/end records of current records by its server’s name.

3)

Using radix sort. Though the space cost will increase, we can sort in O(n) time, thus making time complexity of whole program be O(n + k\* log n)

4)

Sort the k queries. With sorted queries, we only search the list of lasting times once (liner search), therefore, the total search time is O(n) rather than O(k\*log n). Then the total time complexity is O(n\*log n + k\*log k + n)=O(n\*log n + k\*log k). In addition, if we sort by radix ( in 3) ), the total time complexity would be O(n + k + n)=O(n + k).

However, solutions in 3) and 4) may not be practical for the great space cost.

### Appendix: Source Code (in C)

1. /\*ArrangementTasks.c\*/
2. /\*headers\*/
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <assert.h>
6. #include <string.h>
8. /\*Macro Constant\*/
10. /\*
11. \* CUTOFF:
12. \* when function QuickSort is called,
13. \* if the size of the array to sort is
14. \* less than CUTOFF, we use insertion
15. \* sort instead  of quick sort.
16. \*/
17. #define CUTOFF 15
19. #define SERVERNAMECOUNTS 7 //the length of each server's name.
21. #define TRUE 1
22. #define FALSE 0
24. /\*Macro Functions\*/
25. #define New(type,n) malloc(sizeof(type)\*(n))            //alloc a space with specific size
26. #define MAX(a,b) (((a) > (b)) ? (a) : (b))
27. #define MIN(a,b) (((a) < (b)) ? (a) : (b))
28. #define MAXSIZE MAX(sizeof(int\*),sizeof(int))
30. /\*
31. \* lea:
32. \* Load Effective Address,
33. \* that is, returning an
34. \* address of the memory.
35. \*
36. \* lea(base,offset) = &base[offset]
37. \* with a specific type, the size
38. \* of which is block\_size
39. \*/
40. #define lea(base,offset) ((base)+(offset)\*(block\_size))
42. /\*
43. \* record:
44. \* records the start or
45. \* end time of a server
46. \*/
47. **typedef** **struct** record{
48. **char** server[8];
49. **int** time;
50. }\*record;
52. /\*
53. \* interval:
54. \* records the start time
55. \* and end time of a server.
56. \* we don't save the name of
57. \* the server.
58. \*/
59. **typedef** **struct** interval{
60. **int** begin,end;
61. }\*interval;
63. /\*
64. \* times:
65. \*
66. \* times\_list:
67. \* records all the continuous
68. \* running times available for
69. \* tasks.
70. \*
71. \* times\_sum\_list:
72. \* times\_sum\_list[i] = sum([times\_list[i] for i in range(i,len)])
73. \*
74. \* len:
75. \* the length of list
76. \*/
77. **typedef** **struct** times{
78. **int**\* times\_list;
79. **int**\* times\_sum\_list;
80. **int** len;
81. }\*times;
83. /\*
84. \* function CreateIntervals:
85. \* According to the n server records,
86. \* we save the start time and end time
87. \* of each server.
88. \*
89. \* return the array of intervals.
90. \*/
91. interval\* CreateIntervals(record\* records,**int** n);
93. /\*
94. \* function MergeIntervals:
95. \* if a interval overlaps another
96. \* interval, merge them together.
97. \*
98. \* return: times t, which contains
99. \* the sorted information of all
100. \* merged intervals, which are the
101. \* continuous running time available
102. \* for tasks.
103. \*/
104. times MergeIntervals(interval\* intervals,**int** n);
106. /\*
107. \* function NewInterval:
108. \* create a new variable of
109. \* type struct interval, and return
110. \* its pointer.
111. \* function arguments are the two
112. \* attributes of the struct.
113. \*/
114. interval NewInterval(**int** begin,**int** end);
116. /\*
117. \* function NewRecord:
118. \* create a new variable of
119. \* type struct record, and return
120. \* its pointer.
121. \* function arguments are the two
122. \* attributes of the struct.
123. \*/
124. record NewRecord(**char**\* server,**int** time);
126. /\*
127. \* function read\_record:
128. \* read records from the standard
129. \* input.
130. \* n: the number of the records to
131. \* read
132. \*
133. \* return the pointer of arrays of records.
134. \*/
135. record\* read\_record(**int** n);
137. /\*
138. \* function read\_query:
139. \* read a query from the standard
140. \* input.
141. \*
142. \* return the computing time of
143. \* the task.
144. \*/
145. **int** read\_query();
147. /\*
148. \* function cmp\_int:
149. \* Compare two integers.
150. \* arguments: bigger, smaller are
151. \* the addresses of two integers.
152. \*
153. \* If bigger is no less than smaller, return True;
154. \* else return False.
155. \*/
156. **int** cmp\_int(**void**\* bigger,**void**\* smaller);
158. /\*
159. \* function BinarySearch:
160. \* Find the position of an given element if
161. \* it is in the array using binary search.
162. \*
163. \* array[]: The function will search in this array.
164. \* n: the length of the array.
165. \* key: the integer to find.
166. \*
167. \* return the position of key in the array.
168. \*/
169. **int** BinarySearch(**int** array[],**int** n,**int** key);
171. /\*
172. \* function cmp\_start\_time:
173. \* Compare the start time of two intervals.
174. \* arguments: bigger, smaller are the
175. \* addresses of pointers of two intervals.
176. \*
177. \* If bigger is no less than smaller, return True;
178. \* else return False.
179. \*/
180. **int** cmp\_start\_time(**void**\* bigger,**void**\* smaller);
182. /\*
183. \* function cmp\_start\_time:
184. \* Compare the name of two records.
185. \* arguments: bigger, smaller are the
186. \* addresses of pointers of two records.
187. \*
188. \* If bigger is no less than smaller, return True;
189. \* else return False.
190. \*/
191. **int** cmp\_record(**void**\* bigger,**void**\* smaller);
193. /\*
194. \* function QuickSort: (like qsort in stdlib.h)
195. \* Sorting function using
196. \* quick sort algorithm.
197. \* When the array size is
198. \* too short, using insertion
199. \* sort algorithm.
200. \*
201. \* array: the array to sort
202. \* block\_size: the size of each element
203. \* left: the left bound of the array to sort
204. \* right: the right bound of the array to sort
205. \* cmp: the method to compare any two elements
206. \*/
207. **void** QuickSort(**void**\* array,**size\_t** block\_size,**int** left,**int** right,**int**(\*cmp)(**void**\* bigger,**void**\* smaller));
209. /\*
210. \* function median:
211. \* Given an array, compare the
212. \* array[left] and array[right]
213. \* and array[(left+right)/2],and
214. \* find the second largest one,
215. \* exchange it with array[right]
216. \*
217. \* array: the array to sort
218. \* left,right: the first and
219. \* third elements to compare.
220. \*
221. \* cmp: the method to compare any two elements
222. \*/
223. **void** median(unsigned **char**\* head,**size\_t** block\_size,**int** left,**int** right,**int**(\*cmp)(**void**\* bigger,**void**\* smaller));
225. /\*
226. \* function swap:
227. \* exchange the two elements of
228. \* an array.
229. \*
230. \* array: the array of any type
231. \* block\_size: the size of each element
232. \* a,b: the positions of two elements to swap
233. \*/
234. **void** swap(unsigned **char**\* array,**size\_t** block\_size,**int** a,**int** b);
236. /\*
237. \* function InsertionSort:
238. \* Sorting function using
239. \* insertion sort algorithm.
240. \*
241. \* array: the array to sort
242. \* block\_size: the size of each element
243. \* n:the length of the array
244. \* cmp: the method to compare any two elements
245. \*/
246. **void** InsertionSort(**void**\* array,**size\_t** block\_size,**int** n,**int**(\*cmp)(**void**\* bigger,**void**\* smaller));
248. /\*
249. \* function process\_query:
250. \* Given a bances of queries,
251. \* generate a text file with
252. \* output in it, that is, the
253. \* max start points for the task
254. \* of each query.
255. \*
256. \* t: the time information
257. \* len: the number of continuous integers
258. \* queries: the queries put in
259. \* filename: the file to put the output
260. \*/
261. **void** process\_query(times t,**int** len,**int** queries,**char** filename[]);
263. /\*
264. \* function process\_query:
265. \* show a the content of a file
266. \*
267. \* filename: the name of file
268. \*/
269. **void** cat(**char** filename[]);
271. /\*
272. \* function DestoryIntervals:
273. \* Destory the interval struct
274. \* arrays. n is its length.
275. \*/
276. **void** DestoryIntervals(interval\* intervals,**int** n);
278. /\*
279. \* function DestoryRecords:
280. \* Destory the record struct
281. \* arrays. n is its length.
282. \*/
283. **void** DestoryRecords(record\* records,**int** n);
285. **int** main(){
286. //n: the number of server records
287. //k: the number of queries
288. **int** n,k;
290. scanf("%d%d",&n,&k);
292. record\* records=read\_record(n);
294. interval\* intervals=CreateIntervals(records,n/2);
296. DestoryRecords(records,n);
298. times t=MergeIntervals(intervals,n/2);
300. DestoryIntervals(intervals,n/2);
302. process\_query(t,t->len,k,"output");
304. cat("output");
306. **return** 0;
307. }
309. **void** InsertionSort(**void**\* array,**size\_t** block\_size,**int** n,**int**(\*cmp)(**void**\* bigger,**void**\* smaller)){
310. **int** i,j;
311. unsigned **char**\* head= array;// let the pointer of array be the byte type
312. unsigned **char**  tmp[MAXSIZE];
313. **for**(i=1;i<n;i++){
314. memcpy(tmp,lea(head,i),block\_size);// tmp=head[i]
315. **for**(j=i-1;j>=0;j--){
316. **if**((\*cmp)(lea(head,j),tmp)){   //if head[j] > tmp
317. memcpy(lea(head,j+1),lea(head,j),block\_size); //head[j+1]=head[j]
318. }
319. **else** **break**;
320. }
321. memcpy(lea(head,j+1),tmp,block\_size); //head[j+1]=tmp
322. }
323. }
325. **void** swap(unsigned **char**\* array,**size\_t** block\_size,**int** a,**int** b){// swap array[a] and array[b]
326. unsigned **char** tmp[MAXSIZE];
327. memcpy(tmp,lea(array,a),block\_size);
328. memcpy(lea(array,a),lea(array,b),block\_size);
329. memcpy(lea(array,b),tmp,block\_size);
330. }
332. **void** median(unsigned **char**\* head,**size\_t** block\_size,**int** left,**int** right,**int**(\*cmp)(**void**\* bigger,**void**\* smaller)){
333. **int** mid=(left+right)/2;
334. **if**((\*cmp)(lea(head,left),lea(head,right))){// if head[left]>=head[right]
335. **if**((\*cmp)(lea(head,mid),lea(head,right))) // if head[mid]>=head[right]
336. **if**((\*cmp)(lea(head,left),lea(head,mid))) // if head[left]>=head[mid]
337. swap(head,block\_size,right,mid);
338. **else**
339. swap(head,block\_size,right,left);
340. }
341. **else**
342. **if**((\*cmp)(lea(head,right),lea(head,mid))) // if head[right]>=head[mid]
343. **if**((\*cmp)(lea(head,mid),lea(head,left))) // if head[mid]>=head[left]
344. swap(head,block\_size,right,mid);
345. **else**
346. swap(head,block\_size,right,left);
347. }
349. **void** QuickSort(**void**\* array,**size\_t** block\_size,**int** left,**int** right,**int**(\*cmp)(**void**\* bigger,**void**\* smaller)){
350. unsigned **char**\* head=array;
351. **if**(right<left+CUTOFF)// array is too short. use insetion sort.
352. InsertionSort(lea(head,left),block\_size,right-left+1,cmp);
353. **else**{
354. //the element to divide the array into left part and right part is put in the right temporarily.
355. **int** i=left,j=left;//i: the left bound of right part. j: the working pointer
356. median(head,block\_size,left,right,cmp);//find the middle one and put it in the right.
357. **while**(j<right){
358. **if**((\*cmp)(lea(head,right),lea(head,j))){//if head[right]>=head[j]
359. swap(head,block\_size,i,j);
360. i++;
361. }
362. j++;
363. }
364. swap(head,block\_size,i,right);
366. QuickSort(array,block\_size,left,i-1,cmp);
367. QuickSort(array,block\_size,i+1,right,cmp);
368. }
369. }
371. **int** cmp\_record(**void**\* bigger,**void**\* smaller){
372. //first compare two records by name(ascii order), then by time(late>early).
373. record a=\*(record\*)bigger,b=\*(record\*)smaller;
374. **int** flag=strcmp(a->server,b->server);
375. **if**(flag>0 || flag==0 && a->time>b->time)
376. **return** TRUE;
377. **else**
378. **return** FALSE;
379. }
381. **int** cmp\_start\_time(**void**\* bigger,**void**\* smaller){
382. interval a=\*(interval\*)bigger,b=\*(interval\*)smaller;
383. **if**(a->begin>=b->begin)
384. **return** TRUE;
385. **else**
386. **return** FALSE;
387. }
389. **int** cmp\_int(**void**\* bigger,**void**\* smaller){
390. **return** \*(**int**\*)bigger >= \*(**int**\*)smaller;
391. }
393. record NewRecord(**char**\* server,**int** time){
394. record r = New(**struct** record,1);
395. assert(r);
396. strcpy(r->server,server);
397. r->time=time;
398. **return** r;
399. }
401. interval NewInterval(**int** begin,**int** end){
402. interval intrvl=New(**struct** interval,1);
403. assert(intrvl);
404. intrvl->begin=begin;
405. intrvl->end=end;
406. **return** intrvl;
407. }
409. record\* read\_record(**int** n){
410. record\* records = New(record,n);
411. assert(records);
412. **char** server[SERVERNAMECOUNTS+1]; // "+1" is for '\0'
413. **int** time,hh,mm,ss;
414. **while**(n--){// n is declining, being the counter
415. scanf("%s %d:%d:%d",server,&hh,&mm,&ss);
416. time=hh\*3600+mm\*60+ss;
417. records[n]=NewRecord(server,time);
418. }
419. **return** records;
420. }
422. interval\* CreateIntervals(record\* records,**int** n){
423. // sort the records by name and time,
424. // and select each two be the start time
425. // and end time of a interval.
426. // then sort the intervals by their start time
427. **int** i=0;
428. interval\* intervals = New(interval,n);
429. assert(intervals);
430. QuickSort(records,**sizeof**(record),0,2\*n-1,cmp\_record);
431. **for**(i=0;i<n;i++){
432. intervals[i]=NewInterval(records[2\*i]->time,records[2\*i+1]->time);
433. }
434. QuickSort(intervals,**sizeof**(interval),0,n-1,cmp\_start\_time);
435. **return** intervals;
436. }
438. times MergeIntervals(interval\* intervals,**int** n){
440. times t=New(**struct** times,1);
441. assert(t);
442. t->len=0;
443. t->times\_list=New(**int**,n);
444. t->times\_sum\_list=New(**int**,n);
445. assert( t->times\_list && t->times\_sum\_list);
446. **int** begin=intervals[0]->begin,end=intervals[0]->end,i;
448. // there's intervals A and B
449. // If their time order is like following,
450. // Astart --> Bstart --> Aend --> Bend
451. // then we can merge them together.
452. **for**(i=1;i<n;i++){
453. // begin: the start time of current merged(or not yet) interval
454. // end: the end time of current merged(or not yet) interval
455. **if**(intervals[i]->begin>=end){//it is the case: Astart --> Aend --> Bstart, so they can't merge together
456. t->times\_list[t->len++]=end-begin;
457. begin=intervals[i]->begin;
458. end=intervals[i]->end;
459. }
460. **else**{
461. end=MAX(intervals[i]->end,end);// make end-begin more larger
462. }
463. }
464. t->times\_list[t->len++]=end-begin;
466. QuickSort(t->times\_list,**sizeof**(**int**),0,t->len-1,cmp\_int);
468. //the following routines is to get all starting points for a certain query.
469. t->times\_sum\_list[t->len-1]=t->times\_list[t->len-1];
470. **for**(i=t->len-2;i>=0;i--){
471. t->times\_sum\_list[i]=t->times\_sum\_list[i+1]+t->times\_list[i];
472. }
473. **return** t;
474. }
476. **int** BinarySearch(**int** array[],**int** n,**int** key){
477. **int** i=0,j=n-1;
478. **int** mid;
479. **while**(i<=j){
480. mid=(i+j)/2;
481. **if**(array[mid]<key)
482. i=mid+1;
483. **else**
484. j=mid-1;
485. }
486. **return** i;
487. }
489. **int** read\_query(){
490. **int** hh,mm,ss;
491. scanf(" %d:%d:%d",&hh,&mm,&ss);
492. **return** hh\*3600+mm\*60+ss;
493. }
495. **void** DestoryRecords(record\* records,**int** n){
496. **while**(n--) free(records[n]);
497. free(records);
498. }
499. **void** DestoryIntervals(interval\* intervals,**int** n){
500. **while**(n--) free(intervals[n]);
501. free(intervals);
502. }
504. **void** cat(**char** filename[]){
505. **FILE**\* fp=fopen(filename,"r");
506. assert(fp);
507. **char** s[30];
508. **while**(fgets(s,30,fp)) printf("%s",s);
509. fclose(fp);
510. }
512. **void** process\_query(times t,**int** len,**int** querys,**char** filename[]){
513. **FILE**\*fp=fopen(filename,"w");
514. assert(fp);
515. **int** time,i;
516. fprintf(fp,"%d",t->times\_list[len-1]);
517. **while**(querys--){
518. time=read\_query();
519. **if**(time>t->times\_list[len-1]) fprintf(fp,"\n0");// we cannot run this task
520. **else**{
521. i=BinarySearch(t->times\_list,len,time);
522. fprintf(fp,"\n%d",t->times\_sum\_list[i]-(len-i)\*(time-1));// calculate the all staring points for a certain task
523. }
524. }
525. fclose(fp);
526. }

***Declaration:***

***I hereby declare that all the work done in this project titled "*** ***Hard Arrangement of Computation Tasks" is of my independent effort.***