Hard Arrangement of Computation Tasks

朱理真

3190101094

Date: 2021-01-02

Index

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

```c
typedef struct record{
 char server[8];
 int time;
record;
```

interval:

```
* records the start time and end time of a server.
* we don't save the name of the server.
```

```c
typedef struct interval{
    int begin, end;
}*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
```

```c
typedef struct times{
    int* times_list;
    int* times_sum_list;
    int len;
}*times;
```

Main Algorithm

main sturcture:

```
Read the server's records from input

Create structure intervals from records

Merge intervals into structure times

process each query

give output
```

Key Algorithm

Quick Sort

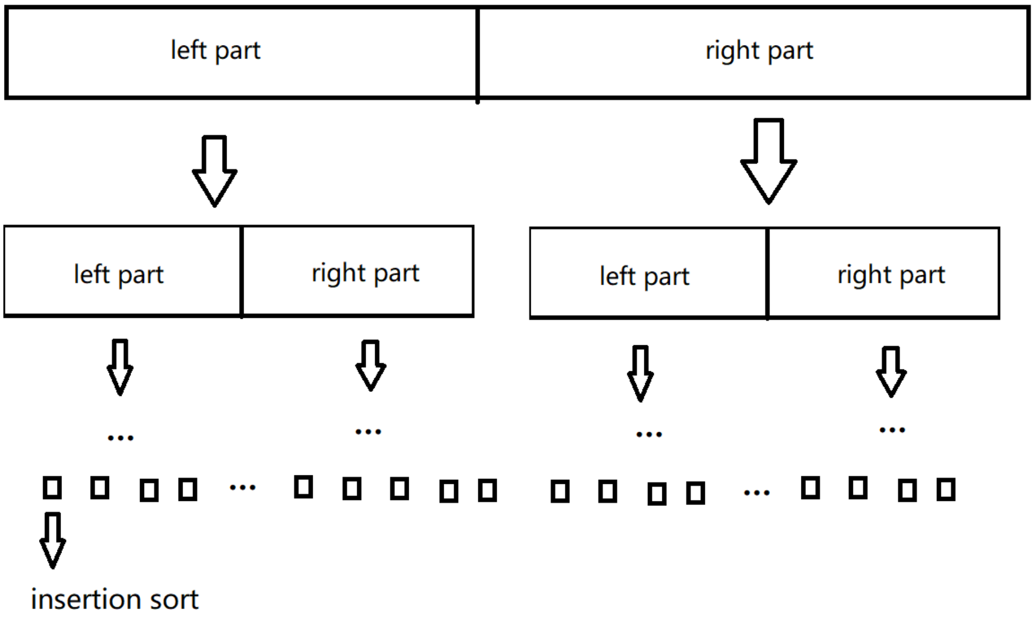
```
Quick Sort(array, left, right):
 If right - left is too short
     Use insertion sort instead

```

```
Compare the left, middle and the right element of the array,
Find the second largest one,
And set it as the element dividing the array into left and right part

Divide the array into left and right part

sort the left part
sort the right part
```

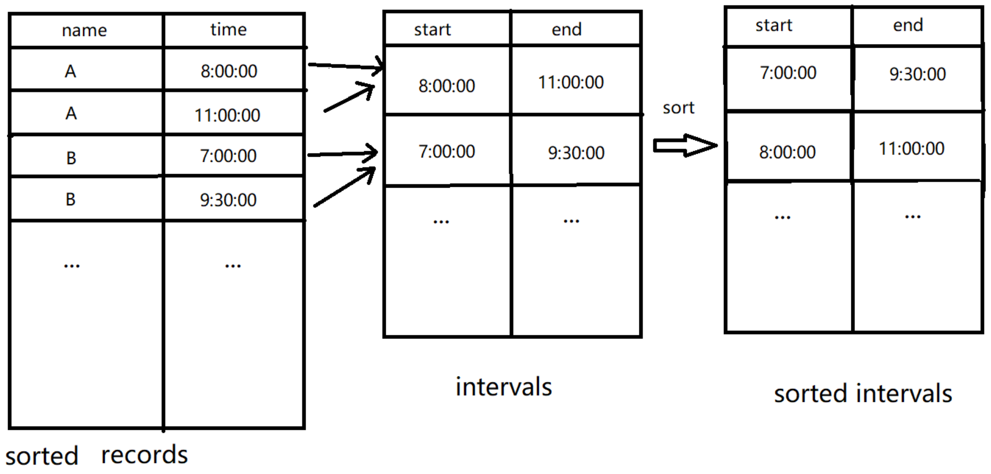


quick sort

Create Intervals

```
Create Intervals(records):
    sort records first by their names, then(if equal) by their time.
    for each two records with same server's name
        create an interval
        let the earlier time of two records be the starting time
           of the interval, later one be the ending time

    sort the intervals by their starting times
    return intervals;
```



name	time
A	8:00:00
A	11:00:00
B	7:00:00
B	9:30:00
…	…

start	end
8:00:00	11:00:00
7:00:00	9:30:00
…	…

start	end
7:00:00	9:30:00
8:00:00	11:00:00
…	…

Create Intervals

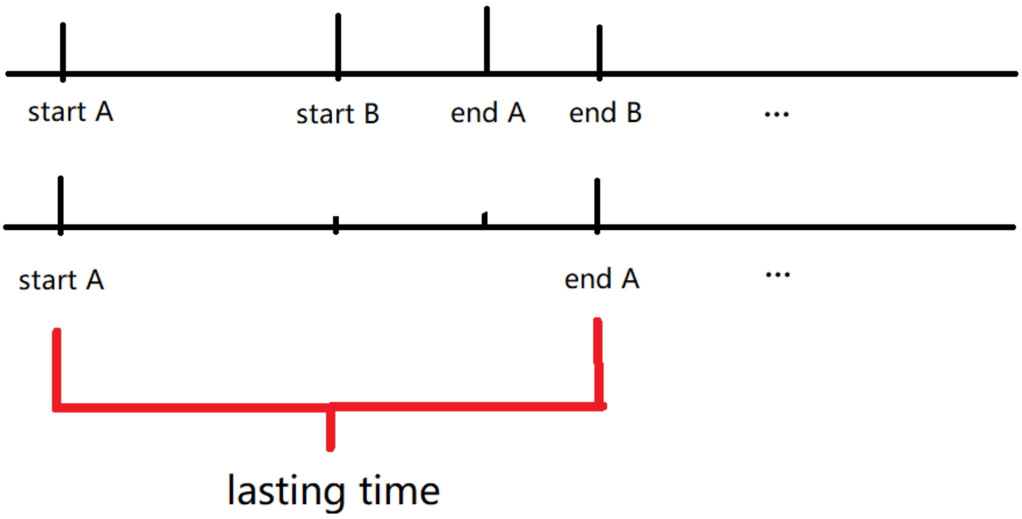
Merge Intervals

```
Merge Intervals(intervals):

    for each two adjacent intervals
        if they overlap, merge them together
        repeat this, until no intervals overlap

    list the lasting times of intervals, call them “time”s

    sort times
```



merge intervals & lasting time

Process Queries

```
Process Queries(times, queries):
```

```
output the longest lasting time.
for each query, get their task’s running time
    for each lasting time larger than the running time
        start time available = lasting time -running time+1
    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.

TestCase 1	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
TestCase 2	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
TestCase	case purpose

3	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
TestCase 4	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
TestCase 5	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
TestCase 6	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

TestCase 7	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
TestCase 8	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
TestCase 9	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
TestCase 10	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) = $\mathbf{O[(k + n)\log n]}$

(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

$$S_{quick-sort} = S_0 = \ c + S_1 = 2c + S_2 = \cdots = kc + S_k$$

where

$$S_k = O(cutoff^2) = O(1)$$

As for k, in the best case,

$$\frac{n}{2^k} = cutoff$$

therefore,

$$k = \Theta\left(\log\frac{n}{cutoff}\right) = \Theta(\log n)$$

and in the worst case,

$$\mathrm{k = O(n)}$$

Hence,

$$S_{quick-sort} = \Theta(\log n)$$

and,

$$S_{quick-sort} = O(n)$$

Above all, S = $6\,O(n) + O(1)$ = $O(n)$

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)

```c
/*ArrangementTasks.c*/
/*headers*/
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

/*Macro Constant*/

/*
 * CUTOFF:
 * when function QuickSort is called,
 * if the size of the array to sort is
 * less than CUTOFF, we use insertion
 * sort instead  of quick sort.
 */
#define CUTOFF 15

```

```
#define SERVERNAMECOUNTS 7 //the length of each server's name.

#define TRUE 1
#define FALSE 0

/*Macro Functions*/
#define New(type,n) malloc(sizeof(type)*(n))            //alloc a space with specific size
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
#define MIN(a,b) (((a) < (b)) ? (a) : (b))
#define MAXSIZE MAX(sizeof(int*),sizeof(int))

/*
 * lea:
 * Load Effective Address,
 * that is, returning an
 * address of the memory.
 *
 * lea(base,offset) = &base[offset]
 * with a specific type, the size
 * of which is block_size
 */
#define lea(base,offset) ((base)+(offset)*(block_size))

/*
 * record:
 * records the start or
 * end time of a server
 */
typedef struct record{
    char server[8];
    int time;
}*record;

/*
 * interval:
 * records the start time
 * and end time of a server.
 * we don't save the name of
 * the server.
 */
typedef struct interval{
    int begin,end;
}*interval;
```

```

/*
 * times:
 *
 * times_list:
 * records all the continuous
 * running times available for
 * tasks.
 *
 * times_sum_list:
 * times_sum_list[i] = sum([times_list[i] for i in range(i,len)])
 *
 * len:
 * the length of list
 */
typedef struct times{
    int* times_list;
    int* times_sum_list;
    int len;
}*times;

/*
 * function CreateIntervals:
 * According to the n server records,
 * we save the start time and end time
 * of each server.
 *
 * return the array of intervals.
 */
interval* CreateIntervals(record* records,int n);

/*
 * function MergeIntervals:
 * if a interval overlaps another
 * interval, merge them together.
 *
 * return: times t, which contains
 * the sorted information of all
 * merged intervals, which are the
 * continuous running time available
 * for tasks.
 */
times MergeIntervals(interval* intervals,int n);

```

```c
/*
 * function NewInterval:
 * create a new variable of
 * type struct interval, and return
 * its pointer.
 * function arguments are the two
 * attributes of the struct.
 */
interval NewInterval(int begin,int end);

/*
 * function NewRecord:
 * create a new variable of
 * type struct record, and return
 * its pointer.
 * function arguments are the two
 * attributes of the struct.
 */
record NewRecord(char* server,int time);

/*
 * function read_record:
 * read records from the standard
 * input.
 * n: the number of the records to
 * read
 *
 * return the pointer of arrays of records.
 */
record* read_record(int n);

/*
 * function read_query:
 * read a query from the standard
 * input.
 *
 * return the computing time of
 * the task.
 */
int read_query();

/*
 * function cmp_int:
 * Compare two integers.
 * arguments: bigger, smaller are
 * the addresses of two integers.
 *
 * If bigger is no less than smaller, return True;
 * else return False.
 */
int cmp_int(void* bigger,void* smaller);

/*
 * function BinarySearch:
 * Find the position of an given element if
 * it is in the array using binary search.
 *
 * array[]: The function will search in this array.
 * n: the length of the array.
 * key: the integer to find.
 *
 * return the position of key in the array.
 */
int BinarySearch(int array[],int n,int key);

/*
 * function cmp_start_time:
 * Compare the start time of two intervals.
 * arguments: bigger, smaller are the
 * addresses of pointers of two intervals.
 *
 * If bigger is no less than smaller, return True;
 * else return False.
 */
int cmp_start_time(void* bigger,void* smaller);

/*
 * function cmp_start_time:
 * Compare the name of two records.
 * arguments: bigger, smaller are the
 * addresses of pointers of two records.
 *
 * If bigger is no less than smaller, return True;
 * else return False.
 */
int cmp_record(void* bigger,void* smaller);

/*
```

```
 * function QuickSort: (like qsort in stdlib.h)
 * Sorting function using
 * quick sort algorithm.
 * When the array size is
 * too short, using insertion
 * sort algorithm.
 *
 * array: the array to sort
 * block_size: the size of each element
 * left: the left bound of the array to sort
 * right: the right bound of the array to sort
 * cmp: the method to compare any two elements
 */
void QuickSort(void* array,size_t block_size,int left,int right,int(*cmp)(void* bigger,void* smaller));

/*
 * function median:
 * Given an array, compare the
 * array[left] and array[right]
 * and array[(left+right)/2],and
 * find the second largest one,
 * exchange it with array[right]
 *
 * array: the array to sort
 * left,right: the first and
 * third elements to compare.
 *
 * cmp: the method to compare any two elements
 */
void median(unsigned char* head,size_t block_size,int left,int right,int(*cmp)(void* bigger,void* smaller));

/*
 * function swap:
 * exchange the two elements of
 * an array.
 *
 * array: the array of any type
 * block_size: the size of each element
 * a,b: the positions of two elements to swap
 */
void swap(unsigned char* array,size_t block_size,int a,int b);

```

```
/*
 * function InsertionSort:
 * Sorting function using
 * insertion sort algorithm.
 *
 * array: the array to sort
 * block_size: the size of each element
 * n:the length of the array
 * cmp: the method to compare any two elements
 */
void InsertionSort(void* array,size_t block_size,int n,int(*cmp)(void* bigger,void* smaller));

/*
 * function process_query:
 * Given a bances of queries,
 * generate a text file with
 * output in it, that is, the
 * max start points for the task
 * of each query.
 *
 * t: the time information
 * len: the number of continuous integers
 * queries: the queries put in
 * filename: the file to put the output
 */
void process_query(times t,int len,int queries,char filename[]);

/*
 * function process_query:
 * show a the content of a file
 *
 * filename: the name of file
 */
void cat(char filename[]);

/*
 * function DestoryIntervals:
 * Destory the interval struct
 * arrays. n is its length.
 */
void DestoryIntervals(interval* intervals,int n);

/*
```

```
 * function DestoryRecords:
 * Destory the record struct
 * arrays. n is its length.
 */
void DestoryRecords(record* records,int n);

int main(){
    //n: the number of server records
    //k: the number of queries
    int n,k;

    scanf("%d%d",&n,&k);

    record* records=read_record(n);

    interval* intervals=CreateIntervals(records,n/2);

    DestoryRecords(records,n);

    times t=MergeIntervals(intervals,n/2);

    DestoryIntervals(intervals,n/2);

    process_query(t,t->len,k,"output");

    cat("output");

    return 0;
}

void InsertionSort(void* array,size_t block_size,int n,int(*cmp)(void* bigger,void* smaller)){
    int i,j;
    unsigned char* head= array;// let the pointer of array be the byte type
    unsigned char  tmp[MAXSIZE];
    for(i=1;i<n;i++){
        memcpy(tmp,lea(head,i),block_size);// tmp=head[i]
        for(j=i-1;j>=0;j--){
            if((*cmp)(lea(head,j),tmp)){   //if head[j] > tmp
                memcpy(lea(head,j+1),lea(head,j),block_size); //head[j+1]=head[j]
            }
            else break;
        }
        memcpy(lea(head,j+1),tmp,block_size); //head[j+1]=tmp
    }
}

void swap(unsigned char* array,size_t block_size,int a,int b){// swap array[a] and array[b]
    unsigned char tmp[MAXSIZE];
    memcpy(tmp,lea(array,a),block_size);
    memcpy(lea(array,a),lea(array,b),block_size);
    memcpy(lea(array,b),tmp,block_size);
}

void median(unsigned char* head,size_t block_size,int left,int right,int(*cmp)(void* bigger,void* smaller)){
    int mid=(left+right)/2;
    if((*cmp)(lea(head,left),lea(head,right))){// if head[left]>=head[right]
        if((*cmp)(lea(head,mid),lea(head,right))) // if head[mid]>=head[right]
            if((*cmp)(lea(head,left),lea(head,mid))) // if head[left]>=head[mid]
                swap(head,block_size,right,mid);
            else
                swap(head,block_size,right,left);
    }
    else
        if((*cmp)(lea(head,right),lea(head,mid))) // if head[right]>=head[mid]
            if((*cmp)(lea(head,mid),lea(head,left))) // if head[mid]>=head[left]
                swap(head,block_size,right,mid);
            else
                swap(head,block_size,right,left);
}

void QuickSort(void* array,size_t block_size,int left,int right,int(*cmp)(void* bigger,void* smaller)){
    unsigned char* head=array;
    if(right<left+CUTOFF)// array is too short. use insetion sort.
        InsertionSort(lea(head,left),block_size,right-left+1,cmp);
    else{
        //the element to divide the array into left part and right part is put in the right temporarily.
```

```c
        int i=left,j=left;//i: the left bound of right part. j: the working pointer
        median(head,block_size,left,right,cmp);//find the middle one and put it in the right.
        while(j<right){
            if((*cmp)(lea(head,right),lea(head,j))){//if head[right]>=head[j]
                swap(head,block_size,i,j);
                i++;
            }
            j++;
        }
        swap(head,block_size,i,right);

        QuickSort(array,block_size,left,i-1,cmp);
        QuickSort(array,block_size,i+1,right,cmp);
    }
}

int cmp_record(void* bigger,void* smaller){
    //first compare two records by name(ascii order), then by time(late>early).
    record a=*(record*)bigger,b=*(record*)smaller;
    int flag=strcmp(a->server,b->server);
    if(flag>0 || flag==0 && a->time>b->time)
        return TRUE;
    else
        return FALSE;
}

int cmp_start_time(void* bigger,void* smaller){
    interval a=*(interval*)bigger,b=*(interval*)smaller;
    if(a->begin>=b->begin)
        return TRUE;
    else
        return FALSE;
}

int cmp_int(void* bigger,void* smaller){
    return *(int*)bigger >= *(int*)smaller;
}

record NewRecord(char* server,int time){
    record r = New(struct record,1);
    assert(r);
    strcpy(r->server,server);
    r->time=time;
    return r;
}

interval NewInterval(int begin,int end){
    interval intrvl=New(struct interval,1);
    assert(intrvl);
    intrvl->begin=begin;
    intrvl->end=end;
    return intrvl;
}

record* read_record(int n){
    record* records = New(record,n);
    assert(records);
    char server[SERVERNAMECOUNTS+1]; // "+1" is for '\0'
    int time,hh,mm,ss;
    while(n--){// n is declining, being the counter
        scanf("%s %d:%d:%d",server,&hh,&mm,&ss);
        time=hh*3600+mm*60+ss;
        records[n]=NewRecord(server,time);
    }
    return records;
}

interval* CreateIntervals(record* records,int n){
    // sort the records by name and time,
    // and select each two be the start time
    // and end time of a interval.
    // then sort the intervals by their start time
    int i=0;
    interval* intervals = New(interval,n);
    assert(intervals);
    QuickSort(records,sizeof(record),0,2*n-1,cmp_record);
    for(i=0;i<n;i++){
        intervals[i]=NewInterval(records[2*i]->time,records[2*i+1]->time);

    }
    QuickSort(intervals,sizeof(interval),0,n-1,cmp_start_time);
    return intervals;
}

```

```c
times MergeIntervals(interval* intervals,int n){

    times t=New(struct times,1);
    assert(t);
    t->len=0;
    t->times_list=New(int,n);
    t->times_sum_list=New(int,n);
    assert( t->times_list && t->times_sum_list);
    int begin=intervals[0]->begin,end=intervals[0]->end,i;

    // there's intervals A and B
    // If their time order is like following,
    // Astart --> Bstart --> Aend --> Bend
    // then we can merge them together.
    for(i=1;i<n;i++){
        // begin: the start time of current merged(or not yet) interval
        // end: the end time of current merged(or not yet) interval
        if(intervals[i]->begin>=end){//it is the case: Astart --> Aend --> Bstart, so they can't merge together
            t->times_list[t->len++]=end-begin;
            begin=intervals[i]->begin;
            end=intervals[i]->end;
        }
        else{
            end=MAX(intervals[i]->end,end);// make end-begin more larger
        }
    }
    t->times_list[t->len++]=end-begin;

    QuickSort(t->times_list,sizeof(int),0,t->len-1,cmp_int);

    //the following routines is to get all starting points for a certain query.
    t->times_sum_list[t->len-1]=t->times_list[t->len-1];
    for(i=t->len-2;i>=0;i--){
        t->times_sum_list[i]=t->times_sum_list[i+1]+t->times_list[i];
    }
    return t;
}

int BinarySearch(int array[],int n,int key){
    int i=0,j=n-1;
    int mid;
    while(i<=j){
        mid=(i+j)/2;
        if(array[mid]<key)
            i=mid+1;
        else
            j=mid-1;
    }
    return i;
}

int read_query(){
    int hh,mm,ss;
    scanf(" %d:%d:%d",&hh,&mm,&ss);
    return hh*3600+mm*60+ss;
}

void DestoryRecords(record* records,int n){
    while(n--) free(records[n]);
    free(records);
}
void DestoryIntervals(interval* intervals,int n){
    while(n--) free(intervals[n]);
    free(intervals);
}

void cat(char filename[]){
    FILE* fp=fopen(filename,"r");
    assert(fp);
    char s[30];
    while(fgets(s,30,fp)) printf("%s",s);
    fclose(fp);
}

void process_query(times t,int len,int querys,char filename[]){
    FILE*fp=fopen(filename,"w");
    assert(fp);
    int time,i;
    fprintf(fp,"%d",t->times_list[len-1]);
    while(querys--){
        time=read_query();
        if(time>t->times_list[len-1]) fprintf(fp,"\n0");// we cannot run this task
        else{
            i=BinarySearch(t->times_list,len,time);
            fprintf(fp,"\n%d",t->times_sum_list[i]-(len-i)*(time-1));// calculate the all staring points for a certain task
        }
    }
    fclose(fp);
}
```

Declaration:

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