Introduction

Here is part of the introduction from PTA:

> Due to the COVID-19, remote working becomes a daily common routine. In the computing center of a university, there are abundant servers running at different time in a day. Now with all the server information available, you are supposed to tell the longest computation task you could run, and find the total amount of valid starting time for a given computation task, if it is runnable. We assume that a task can only be run continuously to finish all its computation, which means you cannot stop it and restart it again to continue. We also assume that a computation task is allowed to switch from one server to another one at any particular time with no cost, if both servers are active at that time.

The program needs to give the total amount of valid starting time for a given computation task, based one the input records.

Analyzing the sample given, let me give some specifications:

1. The server works at the start record but it doesn't work at the stop record. It means that if the input records of server `abcdefg` are `01:00:00` and `02:00:00`, its working time is `[01:00:00, 02:00:00)` in a mathematical representation, which explains the sample output.
2. The program is not demanded to check the validity of the input record. It means that the program doesn't need to detect whether an input like `24:01:01` is valid or not. Because the *Input Specification* in PTA has guaranteed the validity of the input. It's also easy to add some instructions to check the validity.
3. Since the task can be switched with **no cost**, if the working time of two servers overlaps, they are exactly one server that works longer. Take an example, if a server works from `01:00:00` to `03:00:00` and we have another works from `02:00:00` to `04:00:00`, we **equivalently** have one server working from `01:00:00` to `04:00:00`.
4. If there is no input (N=0), the program outputs 0.

Algorithm Specification

Summary

To solve the problem, I need to apply the quick sort twice.

At first, we need to define some structures. A structure storing the input record(a time point) is needed. `hour`, `minute` and `sec` form a time point and `index` is the index of the server that it belongs to. The definition is as follow:

```c
typedef struct timesc
{
    int hour, minute, sec;  // the time
    int index;              // the index of the server that it belongs to
}time;
```

A structure storing the information of the server is needed. `timearr` is an array storing the time records of the server. `ct` is a parameter in convenience of some functions. `pts` is the number of the records. `name` is a string differentiating different servers. The definition is as follow:

```
typedef struct serve
{
    time *timearr;
    int ct;
    int pts;
    char name[8];
}server;
```

A structure storing the information of a time segment is needed. `start` is the start time of the segment. `endtime` is the end time of the segment. `total` is the total span of the segment. The definition is as follow:

```
typedef struct seg
{
    time start;
    time endtime;
    int total;
}segment;
```

Some global parameters are as follow:

```
time *timept;            // the array storing the original information of time
points
server *servers;         // the array storing the information of  servers
segment *segmt;          // the array storing the information of time segments,
including smaller time pieces
segment *timepiece;      // all the small time pieces that haven't been merged
int *result;             // the array storing the result
int servernum=0, segmentnum=0; // the number of servers & the number of time
segments
```

When inputting the records, the records of one server may be out of order, so the program needs to sort the records of a server at first.

So, to solve the problem, the steps should be:

1. Get the input;
2. Allocate records of each server and sort the records respectively;
3. Merge smaller time segments into larger ones;
4. Calculate the total number of valid time points of each query and output the result;

In pseudo-code:

```
int main()
{
    int M, N;
    scanf ("%d %d", &N, &K);    // get the input
    Initialization();
    for(int i=0; i<N; i++)
    {
        getRecord();
    }
    allocateRecords();

    mergeSegments();

    for(int i=0; i<K; i++)
    {
        getQuery();
        calculate();
    }
    output();

    return 0;
}
```

Except that, some functions are useful and need to be declared before explaining each step.

`switchtosec` is a function returning the time in seconds. It implement simple calculation.

```
int switchtosec(time *timepoint)
{
    return timepoint->hour*3600+timepoint->minute*60+timepoint->sec;
}
```

`cmpnumber` is a function traversing `servers` to find a server by its `number`(a string). `servers` is an array including all the existing servers. If `number` is found then the function returns its index in `servers`, or it returns -1.

```
int cmpnumber(char *number)
{
    int i=0;
    while(i<servernum)
    {
        if(number is equal to servers[i].name)
        {
            return i;
        }
        else
        {
            i++;
        }
    }
    return -1;
}
```

`swap` is a simple function swapping the contents of two parameters. In the program, `ElementType` is exactly `time`.

```
void swap(ElementType *A, ElementType *B)
{
    ElementType temp=*A;
    *A=*B;
    *B=temp;
}
```

`medium3` is a function serving quick sort. It picks the pivot and swap the content of `A[Left]`, `A[Center]` and `A[Right]` accordingly. In the program, `ElementType` is exactly `time`.

```
ElementType medium3( ElementType A[ ], int Left, int Right )
{
    int  Center = ( Left + Right ) / 2;
    if ( A[ Left ] > A[ Center ] )
        Swap( &A[ Left ], &A[ Center ] );
    if ( A[ Left ] > A[ Right ] )
        Swap( &A[ Left ], &A[ Right ] );
    if ( A[ Center ] > A[ Right ] )
        Swap( &A[ Center ], &A[ Right ] );
    /* Invariant: A[ Left ] <= A[ Center ] <= A[ Right ] */
    Swap( &A[ Center ], &A[ Right - 1 ] ); /* Hide pivot */
    /* only need to sort A[ Left + 1 ] … A[ Right - 2 ] */
    return  A[ Right - 1 ];  /* Return pivot */
}
```

`Qsort` is the function implementing quick sort. In the program, `ElementType` is exactly `time`.

```
void  Qsort( ElementType A[ ], int Left, int Right )
{   int  i,  j;
    ElementType  Pivot;
    if ( Left + Cutoff <= Right ) {  /* if the sequence is not too short */
        Pivot = Median3( A, Left, Right );  /* select pivot */
        i = Left;     j = Right – 1;
        for( ; ; ) {
     while ( A[ + +i ] < Pivot ) { }  /* scan from left */
     while ( A[ – –j ] > Pivot ) { }  /* scan from right */
     if ( i < j )
        Swap( &A[ i ], &A[ j ] );  /* adjust partition */
     else     break;  /* partition done */
        }
        Swap( &A[ i ], &A[ Right - 1 ] ); /* restore pivot */
        Qsort( A, Left, i - 1 );      /* recursively sort left part */
        Qsort( A, i + 1, Right );    /* recursively sort right part */
    }  /* end if - the sequence is long */
    else /* do an insertion sort on the short subarray */
        insertion( A + Left, Right - Left + 1 );
}
```

`insertion` is a function implementing insertion sort. It's easy and so omitted.

Since the program needs to implement quick sort twice on different element types, there are functions `swaptp`, `medium3tp` , `Qsorttp` and `insertiontp`. They are function implementing quick sort on the type `segment` and sort by the start time. That is the **only difference**.

Get the Input

The main procedure is the function `addtimept`. The function searches the server by its number at first. If the server exists, modify the necessary information of the server and set the index of the time record. If not, 'register' the server in `servers[]`.

```
void addtimept(char *number, time *timepoint)
{
    int i=0;
    i=the index of the server;
    if(number does not exist)
    {
        create a new server in servers[];
    }
    else
    {
        servers[i].pts++;
        timepoint->index=i;
    }
}
```

With the function and a loop, the program can get all the `N` inputs.

Allocate records of each server and sort the records respectively

The allocation of space is easy. Since `servers[i].pts` records the number of the time records, the allocation is:

```
servers[i].timearr=(time *)malloc(sizeof(struct timesc)*(servers[i].pts+2));
```

The space is a little larger (`servers[i].pts+2`) to avoid overflow.

Since `timept[i].index` records the index of the server and `servers[i].ct` can be used as the index of `servers[i].timearr`. The allocation of time records is:

```
int temp=servers[timept[i].index].ct;
servers[timept[i].index].timearr[temp]=timept[i];
servers[timept[i].index].ct++;
```

Finally, sort each `timearr`:

```
for(each existed server)
{
    Qsort(timearr, 0, pts-1);
}
```

Merge smaller time segments into larger ones

Function `findseg` implements the step.

First of all, put all the paired records into one array `timepiece`. Paired records form a time segment and these segments are smaller. **To differentiate with the larger ones**, I call them **time pieces**. Time pieces are **smaller and not merged**.

Then, sort all the time pieces by their start time, preparing for the merge.

Finally, merge the time pieces that can be merged. In the sorted array, if the start time of `segmt[i]` is earlier than the end time of `segmt[i-1]`, `segmt[i]` and `segmt[i-1]` can be merged into a larger segment absolutely. The sort guarantees the start time of `segmt[i-1]` is earlier than both the start time and the end time of `segmt[i]`, which means there is a overlap and the task can be switched in the overlapped time.

In the end of the function, the longest computation task in seconds is calculated and is stored.

In pseudo-code:

```
void findseg(int N)
{
    int j=0;
    for(each records pair)
    {
        timepiece[j].start=start record;
        timepiece[j].endtime=stop record;
        j++;
    }

    Qsorttp(timepiece, 0, N/2-1);

    for(each time piece)
    {
        if(timepiece[j].start is earlier than timepiece[j].endtime)
        {
            Merge them into one segment;
        }
        else
        {
            Create a new segment;
            Calculate the total span of last segment;
        }
    }
    j = the longest computation task;
    result[0]=j;
}
```

Calculate the total number of valid time points of each query and output the result

It is easy.

`judge` is a function calculating the total number of valid time points of an query. It traverses all the segment and return the **sum** of the number of valid time points in each segment. The number of valid time points is always positive.

In pseudo-code:

```
int judge(time *obj)
{
    int span=the span of obj;
    int re=0;
    for(each segment)
    {
        if(span of segment+1>span)
        {
            re+=span of segment+1-span;
        }
    }
    return re;
}
```

Then in `main` (`result[0]` stores the longest computation task in seconds):

```
for(int i=0; i<K; i++)
{
    time obj=the query;
    result[i+1]=judge(&obj);
}
for(int i=0; i<=K; i++)
{
    printf("%d\n", result[i]);
}
```

Testing Results

Input 1

The sample given by PTA.

```
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
```

Desired output:

```
31801
1201
0
13202
20063
64597
13385
64373
```

Actual output:

```
31801
1201
0
13202
20063
64597
13385
64373
```

Pass

Input 2

No input in a compiler.

```
0 2
```

Desired output:

```
0
```

Actual output:

```
0
```

Pass

Input 3

A case that no merges happen.

```
6 3
aaaaaaa 12:00:00
bbbbbbb 13:00:01
ccccccc 13:00:00
bbbbbbb 11:59:59
ccccccc 15:00:00
aaaaaaa 01:00:00
13:59:59
14:00:00
14:00:01
```

Desired output:

```
50400
2
1
0
```

Actual output:

```
50400
2
1
0
```

Pass

Input 4

No queries.

```
12 0
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
```

Desired output:

```
31801
```

Actual output:

```
31801
```

Pass

Input 5

A case that is generated randomly. The case form just one segment from `00:00:00` to `24:60:37`.

A small mistake results in invalid time like `24:60:37` is generated. But the sample still proves that the program works. The result with the mistake is still predictable.

The input is in `test_case.txt` and the output is in `output.txt` in folder `code`.

Pass

Analysis and Comments

Analysis

Assume that there are *N* records and *M* queries.

- Quicksort

  As learned, the four functions work as a whole for quicksort. The time complexity is $O(NlogN)$.

Quicksort takes no extra space.

  time complexity: $O(NlogN)$
  space complexity: $O(1)$

- `findseg()`

  Putting all the time pieces into one array and merging time pieces takes $O(N)$ time, since there are at most $N/2$ servers and $N/2$ time pieces. Calculating the total number takes $O(N)$ time because there are at most $N/2$ time pieces. The quick sort takes $O(NlogN)$ time.

  The function needs no extra space.

  time complexity: $O(NlogN)$
  space complexity: $O(1)$

- `judge()`

  It traverses all the segments to give the sum and there are at most $N/2$ segments. It takes no extra space.

  time complexity: $O(N)$
  space complexity: $O(1)$

- `switchtosec()`

  It is easy to analyze.

  time complexity: $O(1)$
  space complexity: $O(1)$

- `cmpnumber()`

  It traverses the array and there are at most $N/2$ servers.

time complexity: $O(N)$
space complexity: $O(1)$

- `addtimept()`

It needs `cmpnumber()` to find the server and takes no extra space.

time complexity: $O(N)$
space complexity: $O(1)$

So the time complexity of the program is $O(NlogN+M)$. The records need $O(N)$ space and the array storing the result takes $O(M)$ space,the space complexity is $O(N+M)$.

Comments

As far as I can see, there are two possible improvements:

1. The code is somehow too much and can be simplified.
2. When allocating the time records to each server, using the string but not the index may simplify the code and make the program quicker.

Program

The code of the program is as follow, which can also be obtained in arrangement.cpp:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define Cutoff 20
typedef struct timesc
{
    int hour, minute, sec;  // the time
    int index;              // the index of the server that it belongs to
}time;

typedef struct serve
{
    time *timearr;          // time points
    int ct;
    int pts;                // number of time points
    char name[8];
}server;

typedef struct seg
{
    time start;             // start time
    time endtime;           // end time
    int total;              // total span
}segment;

int switchtosec(time *timepoint);                   // express a time in
seconds
```

```
int cmpnumber(char *number);                        // to find the index of a
server
void addtimept(char *number, time *timepoint);      // add a time point to the
structure of the corresponding server
int medium3(time *array, int left, int right);      // to pick the pivot for
'time' structure
void Qsort(time *array, int left, int right);       // quicksort for 'time'
structure
void insertion(time *array, int length);            // insertion sort for 'time'
structure
void swap(time *A, time *B);                        // swap the content of two
'time' structures
int medium3tp(segment *array, int left, int right); // to pick the pivot for
'segment' structure
void Qsorttp(segment *array, int left, int right);  // quicksort for 'segment'
structure
void insertiontp(segment *array, int length);       // insertion sort for
'segment' structure
void swaptp(segment *A, segment *B);                // swap the content of two
'segment' structures
void findseg(int N);                                // to merge the time pieces
and form the larger segments
int judge(time *obj);                               // give the total number of
valid time points for starting the given computation task

time *timept;           // the array storing the original information of time
points
server *servers;        // the array storing the information of  servers
segment *segmt;         // the array storing the information of time segments,
including smaller time pieces
segment *timepiece;     // all the small time pieces that haven't been merged
int *result;            // the array storing the result
int servernum=0, segmentnum=0; // the number of servers & the number of time
segments

int main()
{
    int N, K;
    scanf("%d %d", &N, &K);
    // allocate space
    servers=(server *)malloc(sizeof(struct serve)*(N/2+2));
    timept=(time *)malloc(sizeof(struct timesc)*(N+1));
    result=(int *)malloc(sizeof(int)*(K+2));
    if(N==0)
    {
        printf("0\n");
        system("pause");
        return 0;
    }
    // clear necessary information of an unused element
    for(int i=0; i<N/2+2; i++)
    {
        servers[i].ct=servers[i].pts=0;
    }
    // get the input
    for(int i=0; i<N; i++)
    {
        char number[8];
        scanf("%s %d:%d:%d", number, &timept[i].hour, &timept[i].minute,
&timept[i].sec);
        addtimept(number, &timept[i]);
    }
    // allocate space for the array of time points of each server
    for(int i=0; i<servernum; i++)
    {
        servers[i].timearr=(time *)malloc(sizeof(struct timesc)*
(servers[i].pts+2));
    }
    // allocate time points
    for(int i=0; i<N; i++)
    {
        int temp=servers[timept[i].index].ct;
        servers[timept[i].index].timearr[temp]=timept[i];
        servers[timept[i].index].ct++;
    }
    // sort the time points of each server to form time pieces
    for(int i=0; i<servernum; i++)
    {
        Qsort(servers[i].timearr, 0, servers[i].pts-1);
    }
    // merge small time pieces into larger segments
    findseg(N);

    for(int i=0; i<K; i++)
    {
        time obj;
        scanf("%d:%d:%d", &obj.hour, &obj.minute, &obj.sec);
        result[i+1]=judge(&obj);
    }
    for(int i=0; i<=K; i++)
    {
        printf("%d\n", result[i]);
    }

    /* for debug
    for(int i=0; i<servernum; i++)
    {
        printf("%s#%02d--", servers[i].name, i);
        for(int j=0; j<servers[i].pts; j++)
        {
            printf("%02d:%02d:%02d--", servers[i].timearr[j].hour,
servers[i].timearr[j].minute, servers[i].timearr[j].sec);
        }
        putchar('\n');
    }
    for(int i=0; i<N/2; i++)
    {
        printf("%d:%d:%d-", timepiece[i].start.hour, timepiece[i].start.minute,
timepiece[i].start.sec);
        printf("%d:%d:%d#\n", timepiece[i].endtime.hour,
timepiece[i].endtime.minute, timepiece[i].endtime.sec);
    }
    for(int i=0; i<=segmentnum; i++)
    {
        printf("%d: st#%d:%d:%d end#%d:%d:%d total#%d\n",
i,segmt[i].start.hour,segmt[i].start.minute,segmt[i].start.sec,segmt[i].endtime.
hour,segmt[i].endtime.minute,segmt[i].endtime.sec,segmt[i].total);
    }*/
    system("pause");
    return 0;
}

int switchtosec(time *timepoint)
{
    return timepoint->hour*3600+timepoint->minute*60+timepoint->sec;    //
simple calculation
}

int cmpnumber(char *number)
{
    int i=0;
    while(i<servernum)                                  // traverse to check if
the server has appeared
    {
        if(!strcmp(number, servers[i].name))            // the server has
appeared before
        {
            return i;
        }
        else
        {
            i++;
        }
    }
    return -1;                                          // a new server
}

void addtimept(char *number, time *timepoint)
{
    int i=0;
    if((i=cmpnumber(number))==-1)       // not an appeared server
    {
        strcpy(servers[servernum].name, number);
        servers[servernum].pts++;       // add one time point
        timepoint->index=servernum;     // the index of the server that the time
point belongs to
        servernum++;                    // a new server appears and increment
the number of servers
    }
    else
    {
        servers[i].pts++;               // add one time point
        timepoint->index=i;             // the index of the server that the time
point belongs to
    }
}

void insertion(time *array, int length)
{
    int i=1, j;
    time temp;
    for(i=1; i<length; i++)
    {
        temp=array[i];                  // the standard for sorting is the time
        for(j=i; j>0 && switchtosec(&temp)<switchtosec(&array[j-1]); j--)
        {
            array[j]=array[j-1];
        }
        array[j]=temp;
    }
    return ;
}

void insertiontp(segment *array, int length)
{
    int i=1, j;
    segment temp;
    for(i=1; i<length; i++)
    {
        temp=array[i];                  // the standard for sort is the start
time of the time segment
        for(j=i; j>0 && switchtosec(&temp.start)<switchtosec(&array[j-1].start);
j--)
        {
            array[j]=array[j-1];
        }
        array[j]=temp;
    }
    return ;
}

void swap(time *A, time *B)
{
    time temp=*A;                       // simple swap
    *A=*B;
    *B=temp;
}

void swaptp(segment *A, segment *B)
{
    segment temp=*A;                    // simple swap
    *A=*B;
    *B=temp;
}

int medium3(time *array, int left, int right)
{
    int center=(left+right)/2;          // to pick the pivot, serving the
function of quick sort
    if(switchtosec(&array[left]) > switchtosec(&array[center]))
    {
        swap(&array[left], &array[center]);
    }
    if(switchtosec(&array[left]) > switchtosec(&array[right]))
    {
        swap(&array[left], &array[right]);
    }
    if(switchtosec(&array[center]) > switchtosec(&array[right]))
    {
        swap(&array[center], &array[right]);
    }
    // array[left] <= array[center] <= array[right]
    swap(&array[center], &array[right-1]);
    return switchtosec(&array[right-1]);
}

int medium3tp(segment *array, int left, int right)
{
    int center=(left+right)/2;          // to pick the pivot, serving the
function of quick sort
    if(switchtosec(&array[left].start) > switchtosec(&array[center].start))
    {
        swaptp(&array[left], &array[center]);
    }
    if(switchtosec(&array[left].start) > switchtosec(&array[right].start))
    {
        swaptp(&array[left], &array[right]);
    }
    if(switchtosec(&array[center].start) > switchtosec(&array[right].start))
    {
        swaptp(&array[center], &array[right]);
    }
    // array[left] <= array[center] <= array[right]
    swaptp(&array[center], &array[right-1]);
    return switchtosec(&array[right-1].start);
}

void Qsort(time *array, int left, int right)
{
    int i, j, pivot;
    if(left+Cutoff<=right)
    {
        pivot=medium3(array, left, right);              // to pick the pivot
        i=left;
        j=right-1;
        for(; ; )                                       // sort the array
according to the pivot
        {
            while(switchtosec(&array[++i]) < pivot);
            while(switchtosec(&array[--j]) > pivot);
            if(i<j)
            {
                swap(&array[i], &array[j]);
            }
            else
            {
                break;
            }
        }
        swap(&array[i], &array[right-1]);
        Qsort(array, left, i-1);
        Qsort(array, i+1, right);
    }
    else                                                // if the array is too
small, then apply insertion sort
    {
        insertion(array+left, right+1-left);
    }
```

```
}

void Qsorttp(segment *array, int left, int right)
{
    int i, j, pivot;
    if(left+Cutoff<=right)
    {
        pivot=medium3tp(array, left, right);            // to pick the pivot
        i=left;
        j=right-1;
        for(; ; )                                       // sort the array
according to the pivot
        {
            while(switchtosec(&array[++i].start) < pivot);
            while(switchtosec(&array[--j].start) > pivot);
            if(i<j)
            {
                swaptp(&array[i], &array[j]);
            }
            else
            {
                break;
            }
        }
        swaptp(&array[i], &array[right-1]);
        Qsorttp(array, left, i-1);
        Qsorttp(array, i+1, right);
    }
    else                                                // if the array is too
small, then apply insertion sort
    {
        insertiontp(array+left, right+1-left);
    }
}

void findseg(int N)
{
    segmt=(segment *)malloc(sizeof(struct seg)*(N/2+1));            // allocate
space
    timepiece=(segment *)malloc(sizeof(struct seg)*(N/2+1));
    int i, j=0;
    for(i=0; i<servernum; i++)
    {
        servers[i].ct=0;
        while(servers[i].ct < servers[i].pts)                       // put all
the time pieces into one array
        {
            timepiece[j].start=servers[i].timearr[servers[i].ct++];
            timepiece[j].endtime=servers[i].timearr[servers[i].ct++];
            j++;
        }
    }
    Qsorttp(timepiece, 0, N/2-1);                                   // sort the
time pieces by the start time
    for(segmentnum=0, j=0; j<N/2; j++)
    {
        if(segmentnum==0 && j==0)                                   // the first
allocation
```

```
        {
            segmt[segmentnum].start=timepiece[j].start;
            segmt[segmentnum].endtime=timepiece[j].endtime;
        }
        else if(switchtosec(&timepiece[j].start) <
switchtosec(&segmt[segmentnum].endtime))
        {
            // since the time pieces have been sorted by the start time, so if
the start time of the current time piece
            // is earlier than the end time of the time segment, the current
time piece and the time segment can be merged
            segmt[segmentnum].endtime=
(switchtosec(&timepiece[j].endtime)>switchtosec(&segmt[segmentnum].endtime)) ?
timepiece[j].endtime : segmt[segmentnum].endtime;
        }
        else
        {
            // a new time segment needs to be created
            segmt[segmentnum].total=switchtosec(&segmt[segmentnum].endtime)-
switchtosec(&segmt[segmentnum].start);
            segmentnum++;
            segmt[segmentnum].start=timepiece[j].start;
            segmt[segmentnum].endtime=timepiece[j].endtime;
        }
    }
    // calculate the time span of the last segment
    segmt[segmentnum].total=switchtosec(&segmt[segmentnum].endtime)-
switchtosec(&segmt[segmentnum].start);
    for(i=0, j=0; i<=segmentnum; i++)                           // store the
result
    {
        if(segmt[i].total>j)
        {
            j=segmt[i].total;
        }
    }
    result[0]=j;
}

int judge(time *obj)
{
    int span=switchtosec(obj);              // calculate the time span of the
query
    int re=0;
    for(int i=0; i<=segmentnum; i++)
    {
        if(segmt[i].total+1>span)           // the span of the query is shorter
than the one of the segment
        {
            re+=segmt[i].total+1-span;      // accumulate valid time points
        }
    }
    return re;
}
```

Declaration

I hereby declare that all the work done in this project titled “Project 3 Report” is of my independent effort.