ZJU-FDS-cx2020-Projects (Normal)

The World's Richest

3180103000 许乐乐

Date:2020-12-27

1. Introduction

Sorting is an important operation in computer programming. Its function is to rearrange an arbitrary sequence of data elements (or records) into an ordered sequence of keywords.

To evaluate a sorting algorithm, we can start from the following aspects:

- Time complexity: the time cost from the initial state of the sequence through the process of sorting shift transformation to the final state.
- Space complexity: the space cost from the initial state of the sequence through the process of sorting shift transformation to the final state.
- Usage scenarios: Different kinds of sorting algorithms are suitable for different kinds of scenarios. Sometimes they need to save space and do not require so much time. On the contrary, sometimes they want to consider more time and do not require so much space. Generally, concrete analysis to concrete issues.
- Stability: In the process of sorting, stable algorithms do not change the relative order of elements' positions. On the contrary, unstable algorithms often change the order.

Project "The World's Richest" aims to find the first M people in a given range of their ages, considering the net worth, the age, the name in turn.

2. Algorithm Specification

2.1 Data Structure

The details of the data structure are as follows.

We use a structure `people` to store a person's information. This structure includes:

- `name`: the name of the person. (string of no more than 8 characters without space)
- `age`: the age of the person. (integer in (0, 200])
- `networth`: the net worth of the person. (integer in [−106,106])

Then the array of structs `s` is used to store the sequence to be arranged.

```
struct people
{
    char name[9];
    int age, networth;
}s[100000];
```

2.2 Algorithm

We specify the algorithm used to find the first M people in a given range of their ages, considering the net worth, the age, the name in turn.

2.2.1 int main()

Algorithm:

int main()

Input:

- `n`: the total number of people. (≤105)
- `k`: the number of queries. (≤103)
- `s`: the list of n people.
- `m`: the maximum number of outputs. (≤100)
- `a`: the lower bound of age.
- `b`: the upper bound of age.

Output:

For each query, first print in a line `Case #X:` where `X` is the query number starting from 1. Then output the M richest people with their ages in the range [`Amin`, `Amax`]. Each person's information occupies a line, in the format

```
Name Age Net_Worth
```

In case no one is found, output `None`.

Main Idea:

Input all the information and store the information of all the persons in the array `s`. Use the `qsort()` function provided in the ANSI C standard and declared in the <stdlib.h> to reorder `s`, among which the comparison function `cmp` would be defined behind. Output the first m elements of `s` which are the desired m richest people.

Pseudo Code:

```
//input
input n,k,s;
//sort
qsort(s, n, sizeof(s[0]), cmp);
//output
for (int i = 1; i <= k; i++) {
    int c = 0;
    input m,a,b;
    printf("Case #%d:\n", i);
    for (int j = 0; j < n; j++)
    {
        //check the age range.
        if (s[j].age >= a && s[j].age <= b)
        {
            printf("%s %d %d\n", s[j].name, s[j].age, s[j].networth);
            c++;
        }
        //the number of output reaches m.
        if (c == m)break;
    }
    //no one is found.
    if (c == 0)printf("None\n");
}
```

2.2.1.1 int cmp(a, b)

Algorithm:

int cmp(const void* a, const void* b)

Input:

- `a`: pointer to the element to compare.
- `b`: pointer to the element to compare.

Output:

Return an integer.

If the return value < 0, the element pointed to by `a` will be placed on the left of the element pointed to by `b`; if the return value > 0, the element pointed to by `a` will be placed on the right of the element pointed to by `b`; if the return value = 0, the order of the elements pointed by `a` and `b` is uncertain.

Main Idea:

Compare the order of two elements, considering the net worth, the age, the name in turn.

Pseudo Code:

```
if (aa->networth == bb->networth)
{
    if (aa->age == bb->age)
        return strcmp(aa->name, bb->name) > 0 ? 1 : -1;
    return aa->age > bb->age ? 1 : -1;
}
return aa->networth > bb->networth ? -1 : 1;
```

3. Testing Results

The following shows some typical test cases for verifying the *The World's Richest* implementation and capturing potential bugs. All test cases *pass*.

1. $Test\ Case\ 1:\ n=12,\ k=4$

   the name is up to 8 characters without space.

   $query\ 1:\ m=4,$ the age and net worth of the 2nd \ 3rd \ 4th one is the same.

   $query\ 2:\ m=4,$ the number of people who meet the age requirements is $< m$.

   $query\ 3:\ m=4,$ the number of people who meet the age requirements is $\geq m$.

   $query\ 4:\ m=1,$ the number of people who meet the age requirements is $= 0$, which means no one is found.

2. $Test\ Case\ 2:\ n=1,\ k=3$

   small size

   $query\ 1:\ m=1\ (m=n),$ the number of people who meet the age requirements is $= m$.

   $query\ 2:\ m=2\ (m>n),$ the number of people who meet the age requirements is $< m$.

   $query\ 3:\ m=1,$ the number of people who meet the age requirements is $= 0$, which means no one is found.

3. $Test\ Case\ 3:\ n=4,\ k=1$

   $query\ 1:\ m=4\ (m=n)$, the net worth of the 2nd \ 3rd one is the same. The age and net worth of 3rd/4th one is the same.

4. $Test\ Case\ 4:\ n=52,\ k=4$

   large size

   $query\ 1:\ m=2\ (large\ n, small\ m)$, the age and net worth of the 1st \ 2nd one is the same.

   $query\ 2:\ m=8$, the age and net worth of the 1st \ 2nd one is the same.

   $query\ 3:\ m=3$, the age is up to 200, the net worth is up to 1000000 and -1000000.

   $query\ 4:\ m=52\ (large\ n, large\ m)$

4. Analysis and Comments

- `qsort()`:

  For small-scale data, `qsort()` takes precedence over `merge_Sort` to sort the input data. For lage-scale data, qsort () will use the `quick_Sort` algorithm instead. In the process of `quick_Sort` , when the number of elements in the range to be sorted ≤ 4, `qsort()` degenerates to `insert_Sort` and does not continue to use recursion for `quick_Sort`.If the recursion is too deep, it will lead to the problem of stack overflow. `qsort()` solves this problem by implementing a stack on the heap and simulating recursion manually. Therefore, the time complexity of `qsort()` is $O(nlogn)$, the space complexity is $O(logn)$, which should be optimal generally.

Because of Traversing each element in the I/O process, the time complexity of input is $O(n)$, and the time complexity of output is $O(m)$. For data storage, the space complexity is $O(n)$.

All in all, the total time complexity is $O(nlogn)$, the total space complexity is $O(n)$.

Appendix: Source Code (in C)

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

/*
* A data structure that stores everyone's information.
*/
struct people
{
    char name[9];
    int age, networth;
}s[100000];

/*
 *    compare the order of two elements.
 * ------------------------------------------------
 *
 *    a: pointer to the element to compare.
 *    b: pointer to the element to compare.
 *
 *    returns:
 *    an integer.
 *    If the return value < 0, the element pointed to by `a` will be
 *    placed on the left of the element pointed to by `b` ; if the
 *    return value > 0, the element pointed to by `a` will be placed
 *    on the right of the element pointed to by `b` ; if the return
 *    value = 0, the order of the elements pointed by `a` and `b` is
 *    uncertain.
 */
int cmp(const void* a, const void* b)
{
    //type conversion.
    struct people* aa = (void*)a, * bb = (void*)b;

    //firstly, compare the net worth.
    if (aa->networth == bb->networth)
    {
        //secondly, compare the age.
        if (aa->age == bb->age)

            //Thirdly, compare the alphabetical order of the names.
            return strcmp(aa->name, bb->name) > 0 ? 1 : -1;

        return aa->age > bb->age ? 1 : -1;
    }
    return aa->networth > bb->networth ? -1 : 1;
}


int main()
{
    int n, k, m, a, b;

    /*
    * input n (≤10•5••) - the total number of people,
    * and k (≤10•3••) - the number of queries.
    */
    scanf("%d %d", &n, &k);

    /*
    * input the name (string of no more than 8 characters without space),
    * age (integer in (0, 200]), and the net worth (integer in [−10•6••,10•6••])
    * of a person.
    */
    for (int i = 0; i < n; i++)
    {
        scanf("%s %d %d", s[i].name, &s[i].age, &s[i].networth);
    }

    /*
    *    void qsort(void *base, size_t nitems, size_t size, int (*compar)(const
void *, const void*))
    *
    *    sort the array.
    * ------------------------------------------------
    *
    *    base: pointer to the first element of the array to sort.
    *    nitems: the number of elements in the array pointed by base.
```

```
     *    size: the size of each element in the array, in bytes.
     *    compar: pointer to the function used to compare two elements.
     *
     *    returns:
     *    void.
     */
    qsort(s, n, sizeof(s[0]), cmp);

    /*
    * deal with k queries.
    */
    for (int i = 1; i <= k; i++)
    {
        int c = 0;

        /*
        * input M (≤100) - the maximum number of outputs,
        * and a、b - [Amin, Amax] which are the range of ages.
        */
        scanf("%d %d %d", &m, &a, &b);

        /* output each query. */
        printf("Case #%d:\n", i);
        for (int j = 0; j < n; j++)
        {
            /* check the age range. */
            if (s[j].age >= a && s[j].age <= b)
            {
                printf("%s %d %d\n", s[j].name, s[j].age, s[j].networth);
                c++;
            }

            /* the number of output reaches m. */
            if (c == m)break;
        }

        /* no one is found. */
        if (c == 0)printf("None\n");
    }
    getchar();
    getchar();
}
```

Declaration

I hereby declare that all the work done in this project is of my independent effort.