The World's Richest

author: A student

date: 2020-12

1. Introduction

@ Forbes magazine publishes every year its list of billionaires based on the annual ranking of the world's wealthiest people. It also gives rankings within a certain range of ages. In this lab project, we use certain sorting strategies to simulate this process and try to optimize it.

Quicksort is the fastest known sorting algorithm in practice. It is a divide and-conquer recursive algorithm. Its average running time is $O(n \log n)$. It is very fast, mainly due to a very tight and highly optimized inner loop. However, when n is relatively small， it may be better to use insertion sort.

Now, given the net worth of *N* people, the goal find the *M* richest people in a given range [`Amin`, `Amax`] of their ages, and print their information in non-increasing order of the net worth. In case there are equal worth, it must be in non-decreasing order of the ages. If both worth and ages are the same, then the output must be in non-decreasing alphabetical order of the names.

2. Algorithm Specification

- **Main data structure 1 :** An array of structures

  Definition : The array of structures is used to store information of each person, including their names, ages and net worth. Each element is defined as

  ```
  struct personInfo {
      char name[9];
      int age;
      int worth;
  }
  ```

- **Main data structure 2 :** An array of Indices.

  Definition : This array is used to store the indices after swapping elements, so that we don't need to swap structures.

- **Algorithm 1** : Quicksort

  Input: $n \geq 1$ unsorted integers

  Output: the same integers as the input, but in an order according to the requirements

  Main Idea: Quicksort is a divide and-conquer recursive algorithm. It picks a *pivot* and partition other numbers into two disjoint groups and then calls itself recursively twice to sort the two resulting subarrays.

  Pseudo Code:

```
procedure Median3(Elementype a[],int left, int right ){
    center = (left + right) /2
    among a[left],a[right],a[center],swap their value according to a given order
    return center
}

procedure quicksort(Elementype a[],int left, int right ){
    Through procedure Median3 :
        sort a[left],a[right],a[center]
```

```
        get the index of the picked pivot -> pivot_p
    set i = left+1, j=right-2
    if(i >= j) return
    else
        swap the value of a[pivot_p] and a[right-1]
    while True do
        while a[i] and a[pivot_p] are in order do
            i++
        end
        while a[j] and a[pivot_p] are in order do
            j--
        end
        if i<j then break
    end
    swap the value of a[i] and a[right-1]
    quicksort(a,left,i-1)
    quicksort(a,i+1,right)
    return
}
```

- **Algorithm 2** : Insertion Sort

  Input: $n \geq 1$ unsorted integers

  Output: the same integers as the input, but in an order according to the requirements

  Main Idea: In Insertion Sort, the array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list. For very small files ($n \leq 20$), quicksort does not perform as well as insertion sort since it is recursive. A common solution to use insertion sort for small files.

  ```
  procedure InsertionSort(Elementype a[],int n ){
      for i=1 to n do
          temp = a[i]
          for j = i to 0 do
              if temp and a[j-1] are not in order then
                  a[j]=a[j-1]
          end
          a[j]=temp
      end
  }
  ```

- **Algorithm 3** : Integration of quick sort and insertion sort

  Input: $n \geq 1$ unsorted integers

  Output: the same integers as the input, but in an order according to the requirements

  Main Idea: For very small files ($n \leq 20$), quicksort does not perform as well as insertion sort since it is recursive. A common solution to use insertion sort for small files. An even better idea is to leave the file slightly unsorted and finish up with insertion sort.

```
procedure sort(Elementype a[],int left, int right ){
    n = right - left
    if n is bigger than a given value then
    quicksort(a,left,right)
    else
        InsertionSort(a+Left,n)
}
```

3. Testing Results

Test cases	Design Purpose	Output	status
1 1 Zoe_Bill 35 2333 1 30 40	Minimum array with a single element	Case #1: Zoe_Bill 35 2333	*pass*
9 4 Zoe_Bill 80 2333 Bob_Volk 35 5888 Anny_Cin 95 999999 Williams 30 -22 Alice 18 5888 Joe_Mike 32 6667 Rosemary 40 5888 Dobby 35 5888 Billy 35 5888 6 15 45 4 70 90 4 15 45 1 45 50	An integrative case: Use insertion sort only 1. check if people of the same net worth is print in non-decreasing order of the ages and check if people of the same net worth and age is print in non-decreasing alphabetical order of the names 2. the case when number of outputs M is larger than number of people within the given age range 3.the case when number of outputs M is smaller than the number of people within the given age range 4.the case when there's no one within the given age range	Case #1: Joe_Mike 32 6667 Alice 18 5888 Billy 35 5888 Bob_Volk 35 5888 Dobby 35 5888 Rosemary 40 5888 Case #2: Zoe_Bill 80 2333 Joe_Mike 32 6667 Alice 18 5888 Billy 35 5888 Bob_Volk 35 5888 Case #4: None	*pass*

Test cases	Design Purpose	Output	status
16 4 Zoe_Bill 80 2333 Bob_Volk 35 5888 Anny_Cin 95 999999 Christ_k 47 2100 Nobody 5 0 Logic_bo 23 390 Ariana_g 25 18900 Mariah 90 4783 Williams 30 -22 Alice 18 5888 Joe_Mike 32 6667 Rosemary 40 5888 Dobby 35 5888 Billy 35 5888 Vander_wo 67 56673 Blake_bl 25 18900 6 15 45 4 70 90 4 15 45 1 100 108	An integrative case: Use both quick sort and insertion sort only 1. check if people of the same net worth is print in non-decreasing order of the ages and check if people of the same net worth and age is print in non-decreasing alphabetical order of the names 2. the case when number of outputs M is larger than number of people within the given age range 3.the case when number of outputs M is smaller than the number of people within the given age range 4.the case when there's no one within the given age range	Case #1: Ariana_g 25 18900 Blake_bl 25 18900 Joe_Mike 32 6667 Alice 18 5888 Billy 35 5888 Bob_Volk 35 5888 Case #2: Mariah 90 4783 Zoe_Bill 80 2333 Case #3: Ariana_g 25 18900 Blake_bl 25 18900 Joe_Mike 32 6667 Alice 18 5888 Case #4: None	*pass*

Table 1 shows some typical test cases for verifying the algorithem.

4. Analysis and Comments

Time complexity analysis

The running time of quicksort is equal to the running time of the two recursive calls plus the linear time spent in the partition:

$$T(n) = T(i) + T(n-i-1) + cn$$

- **Worst-Case :** The pivot is the smallest element all the time, then

$$T(n) = T(n-1) + cn$$

$$T(n) = T(1) + C\sum_{i=1}^{n} i = O(n^2)$$

- **Best-case :** In the best case, the pivot is in the middle

$$T(n) = 2T(\frac{n}{2}) + cn$$
$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c$$
$$\cdots$$
$$\frac{T(n)}{n} = \frac{T(1)}{1} + c\log n$$
$$T(n) = cn\log n + n = O(n\log n)$$

- **Average-Case :** For the average case, we assume that each of the file sizes for S1 is equally likely, and hence has probability 1/n. Then

$$T(n) = \frac{2}{n}[\sum_{j=0}^{n-1} T(j)] + cn$$
$$nT(n) = 2[\sum_{i=0}^{n-1} T(j)] + cn^2 (7.15)$$
$$(n-1)T(n-1) = s[\sum_{i=0}^{n-2} T(j)] + c(n-1)^2 (7.16)$$
$$(7.15) - (7.16) : nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$$
$$\rightarrow \frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$
$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$
$$\cdots$$
$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{n+1} \frac{1}{i}$$
$$\therefore T(n) = O(n\log n)$$

Since the value of net worth and age are varying in a relatively small range($[-10^6, 10^6]$ and $(0, 200]$), it may be better to use bucket sort in some situations, such as a situation when the number of all people is huge, but the number of people of same net worth or same age is relatively small. To be more specific, we can first separate the value of net worth into negative and positive groups, which takes $O(n)$, then sort each group according to net worth first using LSD, which have a time complexity of $O(7 * (n + 10))$. For sorting by ages and names, as only a small amount of people need such comparisons, we can use insertion sort.

Space complexity analysis

For the space requirement, since we need an array to store the n people's information, so the space complexity is $\Theta(n)$

Appendix: Source Code (in C)

```
#include<stdio.h>
#include<string.h>
#define Cutoff 10
```

```c
#define max 100000

//structure for storing information of each person
struct personInfo {
    char name[9];
    int age;
    int worth;
}list[max];

typedef struct personInfo *PI;
void Quicksort(int a[],int N);  //the shell of quicksort
int Median3(int a[],int Left,int Right); // get the pivot
void Qsort(int a[], int Left,int Right);
void swap(int *a,int *b); //exchange two values in the array
void InsertionSort(int a[], int n);
void printcase(int count,int lowbound,int highbound,int *table,int N); //print
the result accroding to each query
int compare(int a, int b); //set up a new standard for comparing elements
according to the requirements

main(){
    int N,K;
    scanf("%d %d",&N,&K);
    int i;
    //store the information into the array of structures
    for(i=0;i<N;i++)
        scanf("%s %d %d",list[i].name,&list[i].age,&list[i].worth);
    //table[N] is used to store the indices of elements after swapping, saving
the time and space to swap structures
    int table[N];
    for(i=0;i<N;i++) table[i]=i;
    Quicksort(table,N);
    int count,lowbound,highbound;
    //for each query, print the results
    for(i=1;i<=K;i++) {
        scanf("%d %d %d",&count,&lowbound,&highbound);
        printf("Case #%d:\n",i);
        printcase(count,lowbound,highbound,table,N);
    }
}

// the shell for quicksort
void Quicksort(int a[],int N){
    Qsort(a,0,N-1);
}

//exchange two values in the array
void swap(int *a,int *b){
    int temp;
    temp=*a;
    *a=(*b);
    *b=temp;
}
// since the results should be ranked not only according to the net worth, but
also ages and names, this function define a new relation.
```

```
// this new relation makes sure that the results are ranked in non-increasing order of the net worths. In case there are equal worths, it must be in non-decreasing order of the ages. If both worths and ages are the same, then the output must be in non-decreasing alphabetical order of the names.
//if it return 1, it means that a and b are in right order, otherwise not.
int compare(int a, int b){
    if(a==b) return 0;
    if (list[a].worth != list[b].worth) return list[a].worth > list[b].worth;
    else if(list[a].age != list[b].age) return list[a].age<list[b].age;
    else return strcmp(list[b].name,list[a].name);
}
// get the pivot and sort a[Left],a[Right] and a[Center]
int Median3(int a[],int Left,int Right){
    int Center = (Left + Right)/2;
    //sort three elements accroding to the given order
    if(compare(a[Center],a[Left])>0) swap(&a[Left],&a[Center]);
    if(compare(a[Right],a[Left])>0) swap(&a[Left],&a[Right]);
    if(compare(a[Right],a[Center])>0) swap(&a[Center],&a[Right]);
    return Center;  //return the position of the pivot in the array
}

//the real quicksort algorithm
void Qsort(int a[], int Left,int Right){
    //if the number of elements to be sorted is less than 1, return directly
    if(Left >= Right) return;
    int i,j;
    int pivotp;
    //if the number of elements is large enough, use quicksort
    if(Left+Cutoff <= Right){
        pivotp=Median3(a,Left,Right);
        i=Left;j=Right-1;
        //the elements in the postion [Left] and [Right -1] are sorted. if i+1>=j-1, the array is already sorted.
        if(i+1>=j-1) return;
        //hide the pivot to the end of the array
        swap(&a[pivotp],&a[Right-1]);
        while(1){
            while(compare(a[++i],a[Right-1])>0) ;  //if a[i] and pivot are in right order,move i forward
            while(compare(a[Right-1],a[--j])>0) ;  //if a[j] and pivot are in right order,move j backward
            if(i<j) swap(&a[i],&a[j]); //as long as i and j do not meet, swap the elements that are in the wrong order
            else break;
        }
        //swap the pivot back to the right place
        swap(&a[i],&a[Right-1]);
        //quicksort the left and right parts that are partitioned based on the pivot
        Qsort(a,Left,i-1);
        Qsort(a,i+1,Right);
    }
    else
        //if the number of elements is relatively small, use insertionsort
        InsertionSort(a+Left,Right-Left+1);
}

void InsertionSort(int a[],int n){
```

```
    int i,j;
    int temp;
    for(i=1;i<n;i++){
        temp=a[i];
        //check the elements before a[i] until the right position for a[i] is
found
        for(j=i;j>0 && compare(temp,a[j-1])>0 ;j--)
            a[j]=a[j-1];
        a[j]=temp;
    }
}

//print the result accroding to each query
void printcase(int count,int lowbound,int highbound, int *table,int N){
    int rc;
    rc=0;
    int i;
    int preage;
    //visit each element in the array to see if it is in the given range of age.
    //use the integer rc to count the number of printed elements
    //if the ith element is in the the given range of age, and rc has not reach
the given number , print it.
    for(i=0;i<N && rc<count;i++){
        preage= list[table[i]].age;
        if(lowbound<=preage && preage<=highbound) {
            printf("%s %d
%d\n",list[table[i]].name,list[table[i]].age,list[table[i]].worth);
            rc++;
        }
    }
    //if no elements satisfy the requirements,print"None"
    if(rc ==0 ) printf("None\n");
}
```

Declaration

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