Solutions for Program Assignment 4

CSCI2100A

March 30, 2015

1 Exercise 4.17

1.1 Analysis

A exhaustive search has the time complexity of O(n^2). By using a hash table, we can reduce it to $O(n)$. The most important thing is to design a hash function. Here the hash function is just to mod some large integer. The following two conditions are recommended: (1) this integer needs to be at least 2 times as large as the data size; (2) this integer needs to be a prime. The last thing is to take care of negative integers since there is no negative index in arrays. The way in the code is to add the size of hash table to the negative remainder (negative integer mod positive integer will have negative remainder).

1.2 Sample code

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#define HASHTABLESIZE 999979

typedef struct
{
    int key[HASHTABLESIZE];
    char state[HASHTABLESIZE];
   /* 0=empty, 1=occupied */
} hashtable;

/* The hash function */
int hash(int input)
{
    int value;
    value = input % HASHTABLESIZE;
    if (value < 0) value = HASHTABLESIZE + value; /*deal with negative input*/
    return value;
}

/* The h function */
int h(int k, int input)
{
    int value;
    value = (hash(input) + k)% HASHTABLESIZE;
    if (value < 0) value = HASHTABLESIZE + value;
```

```
    return value;
}

void insert(int item, hashtable * ht )
{
    int hash_value, i, k;

    hash_value = hash(item);
    i = hash_value;
        k = 1;
        while (ht->state[i]!= 0) {
                i = h(k++,item);
        }
        ht->key[i] = item;
        ht->state[i] = 1;
}

int find(int item, hashtable * ht)
{
    int hash_value, i, k, flag;

    hash_value = hash(item);
    i = hash_value;
        k = 1;
        flag = 0;
        while (ht->state[i]!= 0) {
                if (ht->key[i] == item) {
                        flag = 1;
                        break;
                }
                i = h(k++,item);
        }
        return flag;
}

int main()
{
    int n, m, i, j, num, want, flag;
    int array[100000];
    hashtable *ht;

    scanf("%d%d", &n, &m);
    ht = (hashtable*)malloc(sizeof(hashtable));
    memset(ht->key,0,sizeof(int)*HASHTABLESIZE);
    memset(ht->state,0,sizeof(char)*HASHTABLESIZE);
    for (i=0; i<n; i++) {
        scanf("%d", &num);
        array[i] = num;
        insert(num, ht);
    }
    for (i=0; i<m; i++) {
        scanf("%d", &num);
```

```
        flag = 0;
        for (j=0;j<n;j++) {
            want = num-array[j];
            if (find(want, ht) && (want != array[j])) {
                printf("Yes\n");
                flag = 1;
                break;
            }
        }
        if (!flag) printf("No\n");
    }

    return 0;
}
```

2 Exercise 5.12

2.1 Analysis

A naive way to do this is to use a sorting algorithm whose best time complexity should be $O(n \log n)$. By using a heap, the complexity can be reduced to $O(n \log k)$. Since each pile is sorted. The heap initially has the smallest element of each pile. The size of the heap should be k since there are k piles. When a deletion operation is performed, we need to track the label of pile this deleted element belongs to. Then the next smallest element of that pile will be added into the heap. If one pile is already empty, we reduce the size of the heap by 1.

2.2 Sample Code

```
#include<stdio.h>
#include<limits.h>

struct node{
        int val;
        int listNo;
} heap[1000];

int heapSize;

void Init()
{
        heapSize = 0;
         heap[0].val = -INT_MAX;
         heap[0].listNo = -1;
}

void Insert(int element, int no)
{
        int now;

        heapSize++;
        heap[heapSize].val = element;
        heap[heapSize].listNo = no;
```

```
        now = heapSize;
        while(heap[now/2].val > element)
        {
                heap[now].val = heap[now/2].val;
                heap[now].listNo = heap[now/2].listNo;
                now /= 2;
        }
        heap[now].val = element;
        heap[now].listNo = no;
}

struct node DeleteMin()
{
        struct node minElement,lastElement;
        int child,now;
        minElement = heap[1];
        lastElement = heap[heapSize--];
        /* now refers to the index at which we are now */
        for(now = 1; now*2 <= heapSize ;now = child)
        {
                child = now*2;
                if(child != heapSize && heap[child+1].val < heap[child].val )
                {
                        child++;
                }
                if(lastElement.val > heap[child].val)
                {
                        heap[now].val = heap[child].val;
                        heap[now].listNo = heap[child].listNo;
                }
                else break;
        }
        heap[now] = lastElement;
        return minElement;
}
int main()
{
        int number_of_lists, number_of_elements[1000];
        int i, j, total, element[1000][1000], pointer[1000];
        struct node temp;

        scanf("%d",&number_of_lists);
        total = 0;
        for (i = 0; i<number_of_lists; i++)
        {
                pointer[i] = 0;
                scanf("%d",&number_of_elements[i]);
                total += number_of_elements[i];
                for (j = 0; j<number_of_elements[i]; j++)
                {
                        scanf("%d",&element[i][j]);
```

```
                }
        }

        Init();
        for(i = 0; i<number_of_lists; i++)
        {
                Insert(element[i][pointer[i]], i);
                pointer[i] += 1;
        }

        while (total)
        {
                temp = DeleteMin();
                printf("%d\n", temp.val);
                total -= 1;
                if (pointer[temp.listNo]<number_of_elements[temp.listNo])
                {
                        Insert(element[temp.listNo][pointer[temp.listNo]], temp.listNo);
                        pointer[temp.listNo] += 1;
                }
        }
        return 0;
}
```

3 Exercise 6.10

3.1 Analysis

The solution is based on mergesort. In the merge operation, say, you have two sub-arrays A and B where A is on the left side of B, you need to add the current number of elements in A to the final answer whenever you select an element from B. The reason is that when you select an element from B, it means that this element is smaller than all the remaining elements in A. Each element corresponds to a inversion pair.

3.2 Sample Code

```
#include <stdio.h>

int d[1000000] = {0}; /*temporal array used when merging*/
int s[1000000];
long long ans;

void merg(int left,int right){
        int mid=(left+right)/2;
        int i=left,j=mid+1,k=0;
        while(i<=mid&&j<=right)
        {
                if(s[i]<=s[j])
                {
                        d[k++]=s[i++];
                }
                else
```

```
                {
                        d[k++]=s[j++];
                        ans += mid-i+1;
                }
        }

        while(i<=mid)d[k++]=s[i++];
        while(j<=right)d[k++]=s[j++];
        for(i=left,k=0;i<=right;i++,k++) s[i]=d[k];
}
void mergesort(int left,int right){
        int mid;
        if(left<right){
                mid=(left+right)/2;
                mergesort(left,mid);
                mergesort(mid+1,right);
                merg(left,right);
        }
}

int main()
{
        int cases, total, c, i;

        scanf("%d", &cases);
        for (c = 0; c < cases; c++) {
                ans = 0;
                scanf("%d", &total);
                for(i = 0; i < total; i++)
                {
                        scanf("%d", &s[i]);
                }
                mergesort(0,total-1);
                printf("%lld\n", ans);
        }
        return 0;
}
```