Farhan Zaman 9/24/18
CS21200 10:00AM

HW 2 Report

This homework was just a practice and review of static arrays, working with dynamic arrays 2D arrays, how to pass them as parameters, how to return arrays, syntax of allocating memory for dynamic arrays. I get to work on my problem solving skills a bit more with questions that are asked during interviews.

For question 1 I wrote:

```
int Maxsize=50;
string str="abcdefghijklmnopqrstuvwxyz";

void enterarray(char Ar[], int &size)
{
	char input;
	cout<<" Enter lowercase letters up to 50 and to stop type a number \n";
	do{
		cin>>input;
		Ar[size]=input;
		size++;

	}while(isalpha(input)!=0 && size<=Maxsize);
}
```

This enterarray function takes an already initialized static array and allows user to input up to 50 inputs, while checking if the input is a letter and only a letter. The precondition is that there is space to enter things into the array so size<maxsize. The postcondition is a character array with only letters in it. The worst case time complexity is O(n), because the user can enter n inputs which can be any amount less than maxsize.

For question 2 I wrote:

```
char occurences(char Ar[],int size)
{
	int count[26];
	for(int i=0; i<26; i++)
	{
		count[i]=0;
	}
```

```
	for(int i=0; i<size; i++)
	{
		for(int j=0; j<26; j++)
		{
			if(Ar[i]==str[j])
			{
				count[j]++;

			}
			else
			{
				continue;
			}
		}
	}

	int max=count[0];
	int maxindex=0;
	for(int i=1; i<26; i++)
	{
		if(count[i]>max)
		{
			max=count[i];
			maxindex=i;
		}
	}

	return str[maxindex];

}
```

This occurrences function returns the letter that is most frequently inputted into the array. It does this by keeping track of the amount of every letter inputted in the array, in another array. In the integer array I use a placeholder of max and max index to reserve the spot of the most frequent letter and its first place it was spotted so that I can return it from the str global that holds the position of every letter that corresponds to the index and position of the integer array. The precondition of this is that size<Maxsize, and the array only has letters. The postcondition is that the letter which was most frequent is returned. The worst case time complexity is if size=maxsize, and then if all the letters were inputted and in backwards order and that it's all z's so its O(26n) or O(n).

For question 3 I wrote:

```
void shift(char Ar[], int size)
```

```
{
	cout<<"How many shifts do you want, pos for shifting right, negative for
shifting left \n";
	int rotate;
	cin>>rotate;
	if(rotate>0)
	{
		for(int j=0; j<rotate; j++)
		{
			char temp=Ar[size-1];
			for(int i=size-1; i>0; i--)
			{
				Ar[i]=Ar[i-1];
			}
				Ar[0]=temp;
		}
	}

	if(rotate<0)
	{
		rotate*=-1;
		for(int j=0; j<rotate; j++)
		{
			char temp=Ar[0];
			for(int i=0; i<size-1; i++)
			{
				Ar[i]=Ar[i+1];
			}
				Ar[size-1]=temp;
		}
	}


}
```

This shift function, shifts a 1D array to the left if the input by user is negative, and shifts right if its positive. It does this by storing one corner number and singularly shifting everything down one, and having the other corner take the value of the stored number. It does this for the amount of times the person entered. The precondition is that size<maxsize. The postcondition is that the array is shifted to the left that many times or to the right that many times inputted. The worst case time complexity depends on how many times the user wants to shift the array which can be n so O(n^2);

For question 4 I wrote:

```
char* combine(char Ar1[],int sizen1, char Ar2[],int sizen2)
```

```
{
	char *combine= new char[sizen1+sizen2];
	int i=0, j=0, z=0;
	while(i<sizen1 && j<sizen2)
	{
		if(Ar1[i]<Ar2[j])
		{
			combine[z]=Ar1[i];
			i++;
			z++;
		}
		else
		{
			combine[z]=Ar2[j];
			j++;
			z++;
		}
	}

	while(j<sizen2)
		{
			combine[z]=Ar2[j];
			j++;
			z++;

		}

	while(i<sizen1)
		{
			combine[z]=Ar1[i];
			i++;
			z++;
		}


	return combine;
}
```

The combine function takes 2 sorted arrays and puts them into a dynamic array which has the combined size of the 2 smaller arrays, and is returned. I do this by mimicking the mergesort algorithm and compare the sorted values of each array and manually put it in until one array reaches its size, and then fill up the rest of the combined array with the remainder of the bigger smaller array that we didn't use yet. Since I don't know if the sizes will be equal or different or which one is different, I wrote out both cases and whichever is true will occur. The precondition is that size<maxsize and that the arrays are sorted. The postcondition is that the dynamic array has size of the smaller arrays combined, and contains all elements from both arrays sorted. The

worst case time complexity would be O(2n), which is both smaller arrays being size 50 and the while loop would run 100 max times.

For question 5 I wrote:

```
void entermatrix(char** Ar, int &m, int &n)
{
	cout<<"what do you want the mxn dimensions of the array to be? \n";
	int length,width;
	cin>>length>>width;
	m=length;
	n=width;
	char input;
	for(int i=0; i<m; i++)
	{
		for(int j=0; j<n; j++)
		{
			cout<<"enter the input for position "<<i<<","<<j<<":";
			cin>>input;
			Ar[i][j]=input;
			cout<<endl;
		}
	}
}
```

This entermatrix function takes an initialized dynamic 2D array and allows user to choose its dimensions and input letters only into it. It's parameters are referenced, so in main I can use the dimensions of the matrix to display it. It's precondition is that both its dimensions are less than 50. It's postcondition is a 2D dynamic array is filled with characters and has dimensions m by n. The worst case time complexity would be O(n^2), when both arrays have size 50 or n each.

For question 6 I wrote:

```
void rotate90(char** Ar,int m, int n)
{
	assert(m==n);	//make sure square matrix
	cout<<"Enter number of rotations, pos for counter clockwise, neg for clockwise
\n";
	int rotate;
	cin>>rotate;
	int store=m;
	int store2=m;
	int x=0, y=1;
	if (rotate>0)
	{
```

```
            for(int z=0; z<store/2; z++)           //for the rotations of inner and
outer borders
            {
                  for(int count=0; count<rotate; count++)       //number of
rotations they want corner to corner
                  {
                        for(int count2=0; count2<store2-1; count2++)  //rotate to
the corner for 1 90degree one
                        {
                              char temp=Ar[x][y-1];            //store one corner
as we rotate everything else
                              for(int i=x; i<m; i++)
                              {
                                    Ar[x][i]=Ar[x][i+1];    //shift top row by
one to the left
                              }

                              for(int i=x; i<m; i++)     //right most column
shifted upwads
                              {
                                    Ar[i][m-1]=Ar[i+1][m-1];
                              }

                              for(int i=m-1; i>x; i--)  //shift bottom row to the
right
                              {
                                    Ar[m-1][i]=Ar[m-1][i-1];
                              }

                              for(int i=m-1; i>=y; i--) // right most column
shifted downards
                              {
                                    Ar[i][x]=Ar[i-1][x];
                              }
                              Ar[y][x]=temp;  //shift the corner we stored before
down
                        }
                  }
                  x++;
                  y++;
                  m--;
                  store2=store2-2;
            }
      }
      else if(rotate<0)
      {
            rotate=rotate*-1;
            for(int z=0; z<store/2; z++)
```

```
		{
			for(int count=0; count<rotate; count++)
			{
				for(int count2=0; count2<store2-1; count2++)
				{
					char temp=Ar[x][y-1];
					for(int i=x; i<m; i++)
					{
						Ar[i][x]=Ar[i+1][x];
					}

					for(int i=x; i<m; i++)
					{
						Ar[m-1][i]=Ar[m-1][i+1];
					}

					for(int i=m-1; i>x; i--)
					{
						Ar[i][m-1]=Ar[i-1][m-1];
					}

					for(int i=m-1; i>=y; i--)
					{
						Ar[x][i]=Ar[x][i-1];
					}
					Ar[x][y]=temp;
				}
			}
		}
		x++;
		y++;
		m--;
		store2=store2-2;
	}
}
```

This rotate90 function takes a square matrix and rotates it 90 degrees clockwise is the input is negative, or counterclockwise if the input is positive. It does this by first shifting the outer border of a matrix one at a time. I have a for loop set up so that it will shift size-1 more times so that every corner is rotation 90 at a given time. I have a second for loop for the number of rotations the person wanted, and then finally a 3rd for loop to be able to change indices and rotate the inner square of a matrix if applicable. I rotate the matrix one at a time by using 4 for loops that stores 1 corner temporarily, and starts from there to shifting the letters. The precondition is that it is a square matrix, size<maxsize for both dimensions. The post condition is an array that is rotated 90 degrees a certain number of times and direction based off user input.

[a b c d]
[e f g h]
[i j k l]
[m n o p]

Rotated 1 would be
[d h l p]
[c g k o]
[b f j n]
[a e i m]

It's worst case time complexity would be n^3+4n since there can be 50x50 matrix, and then user wants to rotate it n times so our first 3 floops give n^3 and our 4 for loops shifting it one at a time would be 4n. O(n^3)

For question 7 I wrote:

```
void enlarge(char** Ar,int m, int n)
{
	int size;
	cout<<"What factor do you want to enlarge the matrix \n";
	cin>>size;
	//assert(size>1);
	int bign=n*size;
	int bigm=m*size;
	char** big=new char*[bigm];
	for(int i=0; i<bigm; i++)
	{
		big[i]= new char [bign];
	}
	int bigc=0,bigr=0;

	for(int smallr=0; smallr<m; smallr++)
	{
		bigc=0;
		for(int smallc=0; smallc<n; smallc++)
		{
			for(int countofsmallsize=0; countofsmallsize<size;
countofsmallsize++)
			{
				for(int countofbigsize=0; countofbigsize<size;
countofbigsize++)
				{
					big[bigr][bigc]=Ar[smallr][smallc];
```

```
                                bigc++;
                        }
                bigc-=size;
                bigr++;
                }
        bigr-=size;
        bigc+=size;
        }
bigr+=size;
}

for(int a=0; a<bigm; a++)
{
        for(int b=0; b<bign; b++)
        {
                cout<<big[a][b];
        }
cout<<endl;
}
}
```

For the enlarge function, it takes a 2D dynamic array and enlarges it by a size of user input and outputs an array that acts as such:
Original= [a b c]
[d e f]

Output if user entered 2= [a a b b c c]
[a a b b c c]
[d d e e f f]
[d d e e f f]

I do this by first keeping track of the indexes of the original array and the indexes of the new bigger array. I also keep track of the user input of size and use that to keep track of where the new letter would start. For example if enlargement size was 2, then ever letter will have a 2x2 matrix on their own, and the index of the end would be k*size, where k starts from 0 and goes up to dimension of array/size, which I keep track of with 2 other variables. Everytime all the letters of 1 from the smaller is done, then I reset the indices and continue on with the next new letter. The size of the new array is size*original row and size* original column. Although my algorithm takes a while and the alternative method was doing 1 row, copying it size-1 times and then doing the rest with the others, I wasn't able to figure it out properly. The precondition is that the matrix dimensions are less than maxsize, and that user inputs an enlargement number that is bigger than 1 and is an integer. The postcondition is that an enlarged array is displayed by the size factor of

user input. The worst case time complexity is O(n^4), the 4 nested for loops depended on sizing of original matrix and the user input number as well.

Some improvements I would have for functions, would be the algorithm used to achieve certain tasks. For shifting I know there is a relationship between positive shifts and negative shifts, and I can optimize which route to take to get the same answers. Where size-pos shifts =neg shifts, and whichever number is smaller is the algorithm I follow. Then for sorting since I mimicked mergesort, I think it was the most effective method to go by, but if arrays were initially sorted, I would have an option to sort them then use my algorithm, or input into new array then sort. For the rotate90 function, I think I did it most efficiently by shifting it one at a time, and adding more condition to solve the 90 degree part and the inner part. I think for enlargement, I would have been more efficient doing one row at a time and then copying it, but I didn't know how to go about it.

Here is my code for my header file:

```
#ifndef __ZAMAN_FARHAN_HW2_H__
#define __ZAMAN_FARHAN_HW2_H__

#include <iostream>
#include <cctype>
#include <cassert>
using namespace std;


void enterarray(char Ar[],int &size);
char occurences(char Ar[],int size);
void swap(char x, char y);
void shift(char Ar[], int size);
char* combine(char Ar1[],int sizen1, char Ar2[],int sizen2);
void entermatrix(char** Ar, int &m, int &n);
void rotate90(char** Ar,int m, int n);
void enlarge(char** Ar,int m, int n);

#endif
```