CSCI 2421: Data Structures & Algorithms
Assignment 1

Austin Long

September 20, 2022

Part 1 Question 1

I know a write up is not required for part 1. I am just including a copy of the code that I submitted along with this document for completion.

```
#include <iostream>

int findSum(int a[], int size);

int main()
{
        int n;
        std::cout << "Please enter the size of the array: ";
        std::cin >> n;

        int *array = new int[n];

        std::cout << "Please enter the values of the array: " << std::endl;

        for(int i = 0; i < n; i++)
        {
                std::cin >> array[i];
        }

        std::cout << "The sum of the values in the array is " << findSum(array, n) << s

        return 0;
}

int findSum(int a[], int size)
{
        if(size == 1)
        {
                return a[0];
        }

        return a[size - 1] + findSum(a, size - 1);
}
```

Part 2 Question 1

Problem Statement

Give an efficient solution to determine if there exists an integer i such that $A_i = i$ in an array of integers $A_1 < A_2 < A_3 < ..... < A_n$.

The naive and basic approach is to do the sequential search. Sequentially look at every item in the array and find whether there is a match for $A[i] = i$ for any index i. The worst-case runtime of

this algorithm is O(N), N being the size of input.

Please Note: The array is sorted in ascending order. Can you find a better (more efficient than O(N)) solution to this problem?

Write down the better algorithm (in Pseudocode) and show the worst case and best case run time (using big-oh notation) of the algorithm.

Solution

Algorithm Explanation: Since the array is sorted we can use a binary search method.

In the main function, create a sorted array, initialize a variable to store the size of the array, and a variable to store the value you want to find. Call a function to find the value, returning true or false

Function to FindValue(int val, int[] array, int arrSize)

{

Initialize a variable to store the max and min indicies in the array. Find the middle index based on the max and min values. Loop through the array using the middle index. As long as the min is less than or equal to the max the loop will continue. Each iteration it checks to see if the value at the middle index is equal to the value we want, returns true, otherwise checks if the middle index value is less than or greater than the value we want. If it is less than, the max becomes the mid and if it is greater than the min becomes the mid. A new mid is then calculated based on this new range. If the loop ends without finding the value, it returns false

}

Pseudo Implementation Method:

```
int main()
{
        int[] A = {(A_0, A_1, A_2, ..., A_N)}
        int size = N
        int valueToFind = A_3
        if(FindValue(A_3, A, N))
                print value found
        else
                print value not found
        return 0
}

bool FindValue(int val, int[] A, int N)
{
        int max = N - 1
        int min = 0
        int mid = (max - min) / 2

        while(min <= max)
        {
                if A[mid] == val
                        return true
```

```
                else if A[mid] < val
                        max = mid
                else
                        min = mid
                mid = (max - min) / 2
        }
        return false
}
```

The best case runtime given the algorithm above is O(1) which only occurs when the first middle index used contains the value we are interested in finding.

The worst case runtime given the algorithm above is O(log(N)). The worst case would be finding the value when min = max or not finding the value at all where min > max. This requires us to do a full search of the area, but using the above algorithm the size of the searchable area is cut in half on each iteration leading to a logarithmic time in the worst case scenario. The first iteration leaves you with N/2 searchable area, second iteration is N/4, then N/8, N/16... eventually either finding the value or not finding it.

Part 2 Question 2

Problem Statement

The input is an N by N matrix of numbers. Each individual row is increasing from left to right. Each individual column is increasing from top to bottom. The example of a matrix is given below with N = 3.

$$\begin{bmatrix} 3 & 4 & 5 \\ 4 & 5 & 8 \\ 6 & 7 & 9 \end{bmatrix}$$

The following is the basic naive algorithm (in pseudocode) to search for an item in the matrix:

```
bool search2DMatrix( int a[][], int N)
{
        for(int i = 0; i < N; i++)
              for(int j < 0; j < N; j++)
                    if ( item == a[i][j])
                                return true;
         return false //if not found
}
```

What is the worst-case runtime of this algorithm (in big-oh notation)?

Find an O(N) worst-case algorithm that finds if a number item is in the matrix. Write down the algorithm (in pseudocode) and show the analysis of the runtime of the algorithm.

Hint: Please note that the rows are sorted from left to right and the columns are sorted from top to bottom. You can start from top-right corner of the matrix, With a comparison, either a match is found, you can go left, or go down.

Solution:

The worst-case runtime for the search2DMatrix algorithm shown in the problem statement is $O(N^2)$

Algorithm Explanation

The $O(N)$ solution may be implemented using a binary search method as in question 1 but this time applied to the rows and columns of the 2D matrix. Instead of starting in the middle of the array as in a 1D, we will start in either the top right corner or bottom left corner. This will start us at the index containing the highest row value and lowest column value or the lowest row value and highest column value respectively.

If we start in the bottom left corner, and the value we want to find is greater than the value stored we know we have to move right. If the value is less than, we know we have to move up. Unlike the basic search using the double for loop forcing us to search index by index, this method will use one loop to cycle through the array. Within that loop an if-else statement will check the current index, if the current value is less then the row is shifted up else the column is shifted right. Since only one of these is occuring, each runs in O(N) time since we are only working with NxN matrices and at most we will only have to go through an entire row or an entire column but never both. This loop will run until the current row/col value exceeds the overall size of either. If at any time the value is found it returns true, if the loop ends without finding the value it returns false.

Pseudo Explanation

In the main function, create a sorted 2D array. This is sorted by row increasing from left to right and column increasing from top to bottom. The top left index contains the lowest value in the array and the bottom right value contains the highest value in the array. Initialize a variable to store the size of the array, and a variable to store the value you want to find. Call a function to find the value, returning true or false

Function to FindValue(int val, int[][] array, int rowSize, int colSize)

{

Initialize two variables, one to store the current row and one to store the current column setting them to the index corresponding to the bottom left corner. In this case current row will be rowSize - 1 and current column will be 0. Loop until the current row is less than 0 or until the current column exceeds the column size. While this loops, check the index of the array for an equal value, return true, otherwise check for greater or less than and shift the row up or the column to the right for each respectively. If the loop ends without returning true, return false

}

Pseudo Implementation Method

```
int main()
{
 const int ROW_SIZE = N;
 const int COL_SIZE = N;

 int A[ROW_SIZE][COL_SIZE] =
 {
   {A_00, A_01, A_02, ..., A_0N},
   {A_10, A_11, A_12, ..., A_1N},
   ...
   {A_N0, A_N1, A_N2, ..., A_NN}
 };

 int valueToFind = A_34;

 if(FindValue(A_34, A, ROW_SIZE, COL_SIZE))
 {
  std::cout << "value found";
 }
 else
 {
  std::cout << "value not found";
 }
 return 0;
}

bool FindValue(int val, int A[][N], int rowSize, int colSize)
{
 int currRow = rowSize - 1;
 int currCol = 0;

 while(currRow >= 0 && currCol < colSize)
 {
  if (A[currRow][currCol] == val)
  {
   return true;
  }
  if (A[currRow][currCol] > val)
  {
   currRow--;
  }
  else if(A[currRow][currCol] < val)
  {
   currCol++;
  }
```

```
 }
 return false;
}
```

The above algorithm only works for a sorted array. The worst case scenario is O(N) time because unlike the double for loop, this uses one loop that will only ever go a length of N + N instead of N * N. If the value does not exist in the array, the index tracker will only ever have to move a total combination of the number of rows plus the number of columns. This makes the runtime $O(N) + O(N) = O(N + N) = O(2N) = O(N)$