Edith Cowan University
CSP2348
Data Structures
Assignment 2

Martin Ponce
Student 10371381

Tutor: Jitian Xiao

April 26, 2015

Contents

1 Introduction

This report examines array, singly-linked-list and binary tree data structures and algorithms to interact with them. Implementation of these data structures and algorithms in Java will be outlined in the report, along with analysis of algorithms used.

Throughout this report, it is assumed that the selection of algorithms is based on time efficiency rather than space efficiency. In other words, an algorithm with higher time efficiency but lower space efficiency will be selected over an algorithm with a lower time efficiency and higher space efficiency.

The array data structure will be demonstrated through the implementation of a simple lotto game. The lotto game allows up to 1000 players, with each player picking six unique integers, between 1 and 45. Each player and their respective lotto tickets are represented inside a two-dimensional array, while the winning numbers are represented inside a one-dimensional array. Merge sort is used to sort player tickets and winning numbers arrays, while binary search and an adaptation of an array merge algorithm is used to determine if a player's ticket contains the winning numbers.

2 Arrays

The array data structure is demonstrated through the implementation of a simple lotto game. The lotto game allows up to 1000 players, with each player picking six unique integers, between 1 and 45, which makes up their lotto ticket. Each player and their respective lotto tickets are represented inside a two-dimensional array, while the winning numbers are represented inside a one-dimensional array.

The following classes have been created to represent the lotto game:

- `Main`: The executable class, contains `main()` method
- `PlayerTickets`: Generates a two-dimensional array which represents each player, and their picks for the lotto ticket
- `WinningNumbers`: Generates a one-dimensional array which represents the winning numbers for the lotto game
- `WinningPlayers`: Contains logic to determine who the winning players are
- `Sorter`: Helper class to sort `PlayerTickets` and `WinningNumbers` arrays
- `Randomizer`: Helper class to generate random numbers for each player pick and winning number

2.1 Sorting

In order to use more efficient search algorithms such as binary search, the arrays must be sorted first. The `Sorter` class contains `mergeSort()` which is called to sort elements in an array in ascending order using merge sort algorithm. Implementation is shown in Java code 2.1.

2.1.1 Merge sort algorithm

To sort a[left...right] into ascending order:

```
1. If left <right:
    1.1. Let m be an integer about midway between left and right
    1.2. Sort a[left...m] into ascending order
    1.3. Sort a[m+1...right into ascending order
    1.4. Merge a[left...m] and a[m+1...right] into auxiliary array b
    1.5. Copy all components of b into a[left...right]
2. Terminate
```

(Watt & Brown, 2001, p. 54)

2.1.2 Merge sort Java method

```
 1 private static void mergeSort(int low, int high) {
 2 
 3     // 1.0 If left (low) < right (high)
 4     if(low < high) {
 5 
 6         // 1.1 Let m (mid) be an integer about midway between left and right
 7         int mid = low + (high - low) / 2;
 8 
 9         // 1.2 Sort a[left...m] into ascending order
10         mergeSort(low, mid);
11 
12         // 1.3 Sort a[m+1...right] into ascending order
13         mergeSort(mid + 1, high);
14 
15         // 1.4 Merge a[left...m] and a[m+1...right] into auxiliary array b
16         // call merge() which is O(n)
17         merge(low, mid, high);
18     }
19 }
```

Java code 2.1: Merge sort method

At line 17, the `mergeSort()` method calls supporting method `merge()`, as shown in Java code 2.2, in order to perform step 1.4 of the merge sort algorithm.

```java
private static void merge(int low, int mid, int high) {

    // iterate from low through to high
    for(int i = low; i <= high; i++) {

        // copy each element from the array to sort, to each element into temp array
        mergeTempArray[i] = mergeArrayToSort[i];
    }

    // 1.0 Set i = low, set j = mid + 1, set k = low
    int i = low;
    int j = mid + 1;
    int k = low;

    // 2.0 While i <= mid AND j <= high, repeat:
    while(i <= mid && j <= high) {

        // 2.1 If mergeTempArray[i] <= mergeTempArray[j],
        if(mergeTempArray[i] <= mergeTempArray[j]) {

            // 2.1.1 Copy mergeTempArray[i] into mergeArrayToSort[k], then increment i and k
            mergeArrayToSort[k] = mergeTempArray[i];
            i++;

        // 2.2 If mergeTempArray[i] > mergeTempArray[j],
        } else {

            // 2.2.1 Copy mergeTempArray[j] into mergeArrayToSort[k], then increment j and k
            mergeArrayToSort[k] = mergeTempArray[j];
            j++;
        }
        k++;
    }

    // 3.0 While i <= mid,
    while(i <= mid) {

        // 3.1 Copy mergeTempArray[i] into mergeArrayToSort[k], then increment i and k
        mergeArrayToSort[k] = mergeTempArray[i];
        k++;
        i++;
    }
}
```

Java code 2.2: Merge method

2.1.3 Merge sort analysis

As Watt and Brown (2001, p. 54) explains, analysis of the merge sort algorithm's time complexity involves counting the number of comparisons made during the operation. Let $n = right - left + 1$ be the length of the array, and let $C(n)$ be the total number of comparisons required to sort n values.

Step 1.1 involves dividing the array into two subarrays, $n/2$. The left subarray takes around $C(n/2)$ comparisons to sort, and similarly, the right subarray takes around $C(n/2)$ comparisons to sort.

Step 1.4 involves the merging of each subarray into a sorted array and takes about $n-1$ comparisons to complete. Therefore:

$$C(n) \approx \begin{cases} 2C(n/2) + n - 1 & \text{if n} > 1 \\ 0 & \text{if n} \leq 1 \end{cases} \tag{2.1}$$

Simplifying equation 2.1:

$$C(n) \approx n \times log_2 n \tag{2.2}$$

Therefore the time complexity is $O(n\ log\ n)$.

Space complexity is $O(n)$, since step 1.4 requires an auxiliary array of length n to temporarily store the sorted array.

2.1.4 Merge sort console output

The following console outputs indicate that the merge sort algorithm is functioning correctly.

```
***********************
*** UNSORTED ARRAYS ***
***********************

Player 0001 picks: [16][14][37][31][07][42]
Player 0002 picks: [20][12][26][23][44][16]
Player 0003 picks: [10][32][20][35][02][24]
Player 0004 picks: [06][23][19][35][42][25]
Player 0005 picks: [07][17][28][41][29][38]
Player 0006 picks: [16][05][36][31][04][23]
Player 0007 picks: [20][01][34][37][07][18]
Player 0008 picks: [30][29][10][27][34][05]
Player 0009 picks: [03][27][13][38][28][32]
Player 0010 picks: [08][24][45][14][02][07]

Winning Numbers:   [21][22][10][03][04][13]
```

```
***********************
**** SORTED ARRAYS ****
***********************

Player 0001 picks: [07][14][16][31][37][42]
Player 0002 picks: [12][16][20][23][26][44]
Player 0003 picks: [02][10][20][24][32][35]
Player 0004 picks: [06][19][23][25][35][42]
Player 0005 picks: [07][17][28][29][38][41]
Player 0006 picks: [04][05][16][23][31][36]
Player 0007 picks: [01][07][18][20][34][37]
Player 0008 picks: [05][10][27][29][30][34]
Player 0009 picks: [03][13][27][28][32][38]
Player 0010 picks: [02][07][08][14][24][45]

Winning Numbers:   [03][04][10][13][21][22]
```

2.2 Searching

In order to determine the winners of the lotto game, the program must be able to search each number from the winning numbers array within each array of player tickets. Two sort methods have been selected

References

Watt, D. A., & Brown, D. F. (2001). *Java Collections - An Introduction to Abstract Data Types, Data Structures and Algorithms* (1st ed.). New York: John Wiley & Sons.