CSC 3100 Assignment 2

Ziyu XIE
121090642

October 29, 2022

1 02 Representaion

Since each positive integer n corresponds to a unique binary representation, the first important thing is to represent n in binary form. For this binary number $n_{(2)}$, each bit of it can also be represented as 2 to the b power. If b is not in binary form, we continuously do this and put brackets outside until every digit used to represent this number is in binary form. Namely, this number is a 02 representation.
The pseudo pseudo-code

2-4

Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A. (20points)

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

b. What array with elements from the set {1,2,...,n} has the most inversions? How many does it have?

c. What is the relationship between the running time of insertion sort an the number of inversions in the input array? Justify your answer.

d. Given an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

Solution:
a. $(1, 5); (2, 5); (3, 4); (3, 5); (4, 5)$

b. the array $\{n, n-1, ..., 2, 1\}$ has the most inversions. It has

$$(n-1) + (n-2) + \cdots + (2) + (1) = \frac{[(1) + (n-1)] \cdot (n-1)}{2} = \frac{n(n-1)}{2}$$

c. The bigger the number of inversions is, the longer the running time. Justify: (Here is the code for insertion sort)

```java
import java.util.Arrays;
public class InsertionSort {
    public static void main(String[] args){
        int[] items = {3, 8, 1, 4, 2, 5, 9};
        insertionSort(items);
        System.out.println(Arrays.toString(items));
    }
    private static void insertionSort(int[] items){
        int n = items.length;
        for (int i = 2; i <= n; i++){
            int key = items[i-1];
            int j = i - 1;
            while (j > 0 && items[j-1] > key){
                items[j] = items[j-1];
                j--;
            items[j] = key;
            }
        }
    }
}
```

Look at the "while" loop, since we have set $j = i - 1$ before and $j - -$, we have $j - 1 < i - 1$ is always true in each "for" loop. Then, the condition for the "while" loop to continue is $j - 1 < i - 1$, items[j − 1] > items[i − 1], which means that $(i - 1, j - 1)$ is an inversion.
To be specific, when there is no inversion, we have $T(n) = O(n)$ since we do not go into the while loop; when there is maximun number of inversions, we have $T(n) = O(n^2)$ since we need to go through the while loop.
The bigger the number of the inversions is, the longer the program processing in the "while" loop, the longer the running time.

d. The algorithm is as follows:

```java
import java.util.Arrays;
public class InversionFind {
    public static void main(String[] args){
        int[] items1 = {2,3,8,6,1};
        int[] items2 = {6,5,4,3,2,1};
        System.out.println("Initial: "+Arrays.toString(items1));
        int result1 = inversionFind(items1);
        System.out.println(Arrays.toString(items1));
        System.out.println("The number of inversions one is "+
            result1);
        System.out.println("Initial: "+Arrays.toString(items2));
        int result2 = inversionFind(items2);
        System.out.println(Arrays.toString(items2));
        System.out.println("The number of inversions two is "+
            result2);
    }
    public static int inversionFind(int[] a){
        int[] tmpArray = new int[a.length];
        int result = inversionFindHelp(a, tmpArray, 0,
            a.length-1);
        return result;
    }
    private static int inversionFindHelp(
        int[] a,
        int[] tmpArray,
        int left,
        int right
    ){
        int result = 0;
        if (left < right){
            int center = (left+right) / 2;
            result += inversionFindHelp(a, tmpArray, left,
                center);
            result += inversionFindHelp(a, tmpArray, center+1,
                right);
            result += merge(a, tmpArray, left, center+1, right);
        }
        return result;
    }
    private static int merge(
        int[] a,
        int[] tmpArray,
        int leftPos,
        int rightPos,
        int rightEnd
    ){
        int result = 0;
        int leftEnd = rightPos - 1;
        int tmpPos = leftPos;
        int numElements = rightEnd - leftPos + 1;
        while (
            leftPos <= leftEnd && rightPos <= rightEnd
            ){
                if (a[leftPos] <= a[rightPos]){
                    tmpArray[tmpPos++] = a[leftPos++];
                }
                else {
                    result += (leftEnd-leftPos+1);
                    tmpArray[tmpPos++] = a[rightPos++];
                }
            }
        while (leftPos <= leftEnd){
            tmpArray[tmpPos++] = a[leftPos++];
        }
        while (rightPos <= rightEnd){
            tmpArray[tmpPos++] = a[rightPos++];
        }
        for (int i = 0; i < numElements; i++, rightEnd--){
            a[rightEnd] = tmpArray[rightEnd];
        }
        return result;
    }
}
// the output is:
// Initial: [2, 3, 8, 6, 1]
// [1, 2, 3, 6, 8]
// The number of inversions one is 5
// Initial: [6, 5, 4, 3, 2, 1]
// [1, 2, 3, 4, 5, 6]
// The number of inversions two is 15
```

3.1-5

Prove Theorem 3.1. For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Solution:

If $\Rightarrow$: prove that $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

since $f(n) = \Theta(g(n))$, we have there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$, then we conclude that

(1) there exist positive constants c_2 such that $f(n) \le c_2 g(n)$ for al $n \ge n_0$

(2) there exist positive constants c_1 such that $f(n) \ge c_1 g(n)$ for al $n \ge n_0$

By definition, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Only if$\Leftarrow$: prove that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n)) \Rightarrow f(n) = \Theta(g(n))$

since $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, we have

(1) there exist positive constants c_2 such that $f(n) \leq c_2 g(n)$ for al $n \geq n_0$

(2) there exist positive constants c_1 such that $f(n) \geq c_1 g(n)$ for al $n \geq n_0$

then there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$, By definition, $f(n) = \Theta(g(n))$

3-4.b

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

Solution:
The conjecture is wrong.
For instance, $f(n) = n^3$, $g(n) = n$, then $\Theta(\min(f(n), g(n))) = \Theta(n)$,
which means that $f(n) + g(n) = \Theta(n)$
hence, $f(n) + g(n) = O(n)$
However, $f(n) + g(n) = O(n^3)$, a contradiction.