Quick Sort

Isabel Navarro Guirado

Spring Fall 2022

Introduction

In this assignment we will be looking at a divide and conquer algorithm, the **quick sort** algorithm. The divide and conquer algorithms are algorithms that break down the problem to solve into a collection of sub-problems of smaller size, solve those sub-problems and finally combine the results to obtain a solution for the initial problem.

The **quick sort** algorithm is a recursive algorithm as the sub-problems are of the same type as the original one and are solved using the same technique but they must be fractional in size to the size of the original problem. The sub-problems are generated exclusively from the original problem.

The aim of this assignment is to provide two different implementations of the **quick sort** algorithm : one for arrays and one for linked lists. In each section we would be including the code of each implementation as well as some comments about the time complexity of the algorithm in each of the structure. To end the project, we will do a comparison of the run time of each implementation.

A brief explanation

We will now provide a quick explanation of how the **quick sort** algorithm works. The idea is quite simple; given a collection of elements, we will choose a *pivot* and we will then divide the collection in two sequences, one containing the smallest elements relative to the pivot and one containing the larger elements. Supposing we can sort the two sub-collections separately then we could just combine both sub-collections in one sorted collection. If not, we can choose a *pivot* in each of our sub-collections and repeat the same process. Eventually we will end up with a collection with one or two elements which is easy to sort. We would finish our assignment by combining all the small sub-collections.

When reading the above paragraph, the word *pivot* appears in several occasions but it is unclear which element to choose. This is because it can be any element of the collection. However, the election of the pivot has great influence in the time complexity of the algorithm.

Other sorting algorithms

As mentioned before the quick sort algorithm is a type of divide and conquer algorithm which starts considering the problem of sorting a list with n elements and then divides the problem in sub problems of smaller size.

If we think about other algorithms to sort list, we might realize that merge sort had a similar structure. In merge sort we started with the whole array and we divide it into two and place a sorted version of the first array in one array and a sorted version of the second array in another array and then construct a new array by merging the two arrays.

Merge sort and quick sort are similar types of sorting algorithm as they both belong to the family of the *divide and conquer* algorithms. Quick sort has the advantage that divides the array in a way that all elements in the left of the pivot are smaller than it and all the elements in the right of the pivot are higher than it and therefore the final merge won't be necessary. The price we pay is that we first need to do a partition of the array so that the condition mentioned above holds.

Quick sort can also relate to insertion sort as we

An array implementation

In this section we will be implementing the **quick sort** algorithm for arrays. In order to do so, we will create two functions, the `partition` function and the `sort` function. We will now provide a brief explanation of each of them:

`partition(int[] array, int i, int j) :` This function has three inputs: the array, and two elements which will determine the position of the elements of the array in where we want to do the partition. We will then choose a pivot and divide the array in two sub-arrays, one with the elements that are smaller than the pivot and the other one with the elements that are bigger than the pivot. In this case, we have chose to take as a pivot the element which is at position i in the array (where we want to start the partition). We start looking at the elements of the array from position i. If the element in i is smaller than the pivot, we increment i in one unit and if not, we break the loop and we keep running the program. We do the opposite from position j, if the element in j is bigger than our pivot, then we reduce j in one unit and if not, we stop the loop. If $i \leq j$, the element in i is greater than the element in j, we swap them and repeat the process. If $i > j$, then we swap the pivot with the element in position j and we set the index to j. The function then return an index that will indicate the position of the pivot after doing the partition. All the elements in the left will be smaller and all the items to the right will be bigger than it.

`sort(int[] array, int i, int j) :` This function is a recursive function that will sort our array between from position i to position j. The base case of the recursion occurs when the array has only one item in which case, the function will return the array itself. If the array has more than one item then we use the partition function to obtain an index. All the items that are bigger than our index are in its right and all the items that are in its left are smaller. Three things can occur :

- If the index is i, this is, the first element in the array, then we will only need to sort the right part of the array so we will apply the `sort` function taking as input indices `index+1` and `j`.
- If the index is j, this is, the last element in the array, then we will only need to sort the left part of the array so we will apply the `sort` function taking as input indices `i` and `index-1`.
- Finally, if the index is an element in the middle of the array, different to i and j then we will need to sort the right and left parts of the array separately so we will apply the `sort` function taking as input indices and (`i`, `index-1`) (`index+1`, `j`).

A drawing of the functioning of the **quick sort** algorithm is provided below:

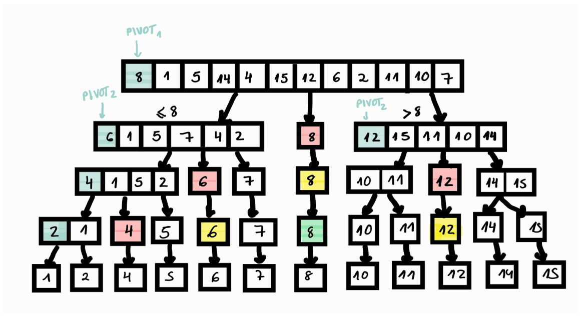

Figure 1: Quick Sort arrays

Code

The code that has been used to implement the **quick sort** algorithm is:

```java
public class quicksort {
    public static int [] swap(int [] array, int i, int j){
        int swap = array[i];
        array[i]=array[j];
        array[j]=swap;
        return  array;
    }
    public static int partition(int[] array, int i, int j) {
        int pivot = array[i];
        int pointer = i;
        int end = j;
        while (pointer != end+1){
            while ((pointer <= end)&&(array[pointer]<=pivot)){
                pointer++;
            }
            while((pointer<=end) && (array[end]>=pivot)){
                end--;
            }
            if (pointer < end){
                swap(array, pointer, end);
                pointer++;
                end--;
            }
        }
        swap(array, i, end);
        int index =end;
        return index;
    }
    public static int [] sort(int[] array, int i, int j){
        if (i == j){ //the array has 1 item
            return array;
        }
        else{
            int index =partition(array, i, j);
            //we have two extreme cases
            if (index==i){
                //if index == 0 it means that all the items in the array
                //are bigger than our pivot and we have to sort that part
                array = sort(array,index+1,j);
            }
            else if (index == j) {
                //if index==array.length-1, all the items in the array are
                //smaller than our pivot, we have to sort that part
                array = sort(array,i,index-1);
            }
            else{
                //if we are not in any of those cases, we sort the array separately
                array = sort(array,i,index-1);
                array = sort(array,index+1,j);
            }
            return array;
        }
    }
}
```

Note. *As in the correction commments it was mentioned that the recursive nature of the* `partition` *function was, apart from clumpsy, the responsible of incorrect results in the comparison with the linked list implementation, a new iterative partition function was created and the previous one was added in the appendix.*

Comments

We then decide to run some benchmarks to figure out the time complexity of the quick sort algorithm. In order to get more accurate results we decided to do the sort operation 100000 times, each time sorting a new array (if we did not create a new array each iteration we will be sorting an already sorted array and the time would not be a good representative).

size	**run time**
100	3000
200	7000
400	16000
800	34799
1600	78199
3200	172100

Table 1: Quick sort Array

From the results in the table we can see that, in average, as we double the size of the array, the execution time is doubled as well and increased by a quantity which becomes bigger with time. For this reason, our first impression is that the time complexity of this algorithm is $O(nLn(n))$. In order to verify our results, we decide to plot the results obtained in the benchmark in the same graph as we plotted the function $f(x) = 5xln(x)$. We could observe that the dots fit quite well and therefore conclude that the quick sort algorithm, when implemented for arrays, has a time complexity of $O(nLn(n))$.

More comments about regarding the time complexity will be made after the linked list implementation.

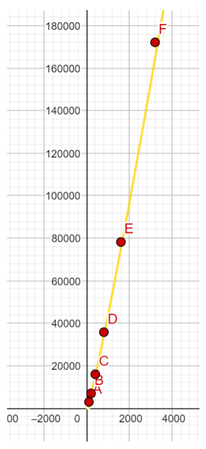

Figure 2: Time complexity - Arrays

Linked list implementation

The implementation for the linked lists is a little bit more complicated as accessing the items of the list is harder. The idea is to take the list and we set our pivot to be the first element. If the list has only one element then we return that element and if not we keep track of the pivot, the current node (now) that starts being the same as the pivot and the next node. We then loop through all the elements in the list and if the element that we are look to is smaller than the pivot, we swap it so that it stays in the right side of the pivot. We keep going like that until we get to the end on the list in which case we, the function will return the element previous to the pivot. We will then have the sort function which fucntions in a similar way as it did in the array implementation and we again consider the three mentioned cases.

Code

We now include the code of the `quicksortlist` class. We will not be including some bits of code like the `Node` class as it has previously been implemented in previous tasks.

```java
public class quicksortlist {
   Node first;
   Node last;
   public static class Node {
   :
   }
   public quicksortlist(){
       first = null;
```

```
    last= null;
}
// we create the append function to add a list in the end of the list
public void append(quicksortlist end){
    if (this.last != null)
        this.last.tail = end.first;
    else
        this.last = end.first;
    this.last = end.last;
}
//we create the prepend function that adds an element at the front of the list
public void prepend (quicksortlist start){
    if (start.last != null)
        start.last.tail=this.first;
    if (this.last == null) //the empty list
        this.last = start.last;
    this.first = start.first;
}
public void cons (Node nd) {
    if (this.last == null) {
        this.first = nd;
    }
    else {
        this.last.tail = nd;
    }
    this.last = nd;
}
public static quicksortlist create (int  n){
    Random rnd = new Random();
    quicksortlist l = new quicksortlist();
    for (int i = 0;i < n; i++){
        l.cons(new Node(rnd.nextInt(2*n)));
    }
    return l;
}
public void show (){
    Node nxt = this.first;
    while (nxt != null){
        System.out.println(nxt.value);
        nxt = nxt.tail;
    }
}

public Node partition (Node beg, Node end){
    if (beg == end || beg == null || end == null)
       return beg;
    Node pivot = beg;
    Node nxt = beg.tail;
    Node now = pivot;
    while(nxt != end.tail){
        if(nxt.value <= pivot.value){
            now = pivot;
            pivot = pivot.tail;
            int temp = nxt.value;
            nxt.value = pivot.value;
            pivot.value = beg.value;
            now.value = temp;
        }
        nxt = nxt.tail;
    }
    return now;
}
public void sort(Node head, Node end){
    if(head == null || head == end){
        return;
    }
    Node previous = partition(head, end);
    if(previous == head && previous != null){
        this.sort(previous.tail, end);
    }else if(previous.tail == end && previous != null){
        this.sort(head, previous);
    }else{
        this.sort(head, previous);
        this.sort(previous.tail.tail, end);
    }
}
```

Comments

As done in previous sections, we now decide to run some benchmarks to figure out the time complexity of the quick sort algorithm when we implement it to sort linked list. We decided run the sort operation in 100000 different linked lists so that the results obtained were more accurate. Some of the results were:

As happened with the results of the benchmark in the array section, as we double the size of the array, the execution time doubles as well and is increased by a quantity that is higher than the one in the array implementation and that gets higher as we increase the size of the array. Our first

size	run time
100	3963
200	8327
400	18305
800	40634
1600	89763
3200	210792

Table 2: Run time - Linked List

impression then is that, as it happened with arrays, the time complexity of the algorithm will be $O(nLn(n))$. We now decide to do a diagram where we plot the run time as a function of the size of the array and we also include the graphic of the function $g(x) = 9xLn(x)$. We can then observe that the dots fit quite well to the graphic which corroborates our initial belief that the quick sort algorithm has a time complexity $O(nLn(n))$ when implemented on the linked list.

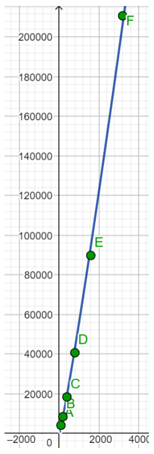

Figure 3: Time complexity - Linked list

Comparison

We will dedicate this section to comparing the results from both implementations.

Note. *As in this assignment I have changed the implementation of the* `partition` *function, results have been altered and therefore, the times have been modified.*

As mentioned before in each section, both arrays have similar execution times and in general cases they have a time complexity of $O(nLn(n))$.

However, there is a slight difference in the time obtained in the array implementation and in the list implementation that becomes more notorious as the size of the array/list grows. To see this difference more explicitly, a table with the results obtained for each data structure is included and accompanied of a graphic were we plot the results of both benchmarks : blue dots for the linked list implementation and the red dots for the array implementation.

size	array	linked list
100	3000	3963
200	7000	8327
400	16200	18305
800	35799	40634
1600	78199	89763
3200	172100	210792

Table 3: Compare

And the mentioned graph that corresponds to the results is:

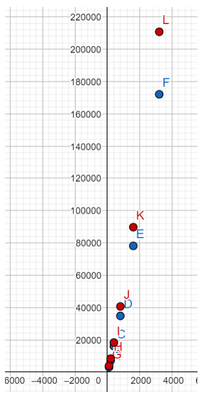

Figure 4: Compare

Worst case scenario

In this section we will be reasoning about the time complexity in the worst case scenario.

As seen in the code, the quick sort algorithm has a recursive nature. Every time we call the `sort` function, we first check that we are sorting more than one element and then we call the `partition` function which will return an integer indicating the location of the pivot element and it then divides the array in two sub arrays and sort them separately. After calling the `partition` function, the pivot element could be in any position of the array.

The worst case scenario would be when the index obtained after applying the `partition` function is an extreme of the array/list as the sub-array will only have one element less than the original array. If this happens, then the algorithm would have a time complexity of $O(n^2)$ because we would be calling the sort function n times (we would be sorting element by element) and in each call we would need to check all the elements remaining. The picture below illustrates what explained in the paragraph:

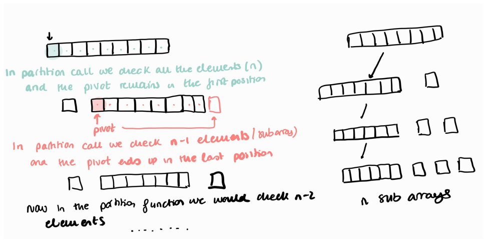
In partition call we check all the elements (n) and the pivot remains in the first position
pivot
In partition call we check n-1 elements (subarray) and the pivot ends up in the last position
Now in the partition function we would check n-2 elements
....
n sub arrays

Figure 5: Caption

Normally, the situation described above does not happen and the partition function returns an element in the middle of the array. If this happens, then the complexity of sorting a list of n elements, as seen in the benchmarks would be $O(nLn(n))$.

Appendix

Partition

```
public static int partition(int[] array, int i, int j){
        int org_i = i;
        int org_j = j;
        int pivot = array[i];
        int index = i;
        while (i<array.length && array[i] <= pivot ){
            i++;
        }
        while(j>0 && array[j]>pivot){
            j--;
        }
        if (i <= j){
            array=swap(array,i,j);
            index = partition(array, org_i, org_j);
        }
        else{
```

```
        array = swap(array, org_i,j);
        index = j;
    }
    return index;
}
```