COMPARISON OF RECURSIVE AND ITERATIVE METHODS USING PROGRAMMING THEOREMS

Tamás Gábor Félegyházi; college student; Dennis Gabor College; felegyhazi.tamas@gmail.com

Ákos Kovács; college student; Dennis Gabor College; akoskovacs993@gmail.com

Sándor Kaczur; assistant lecturer; Dennis Gabor College; kaczur@gdf.hu

Keywords: programming, effectiveness, algorithms, recursion, programming theorems

1. ABSTRACT

In mathematics recursion is a well-known concept and an applied procedure, several definitions and formulae are laid down in a recursive form, and it is the principal base of several theorem's verification. The fast paced advance of the information technology opened countless new opportunities for the appliance of using recursion, complicated operations may be algorithmized in a very simple way with recursion. For example, the exploration of a share on a file server (all folders, and subfolders) can be implemented with a very straightforward algorithm, however, accomplishing the same with nested iterations will result in a lengthy, perspicuous and hard to expand source code [1-4].

Several known examples exist where the problem can be most effortlessly described, understood and solved by reducing the problem to the most trivial base-event. For example: Fibonacci sequence, or the towers of Hanoi, where the base event could be the replacement of all the disks towering above our current disk, and placing all of them (in the same order) to the next rod.

Separating recursive problems into two is practical: in the first set for the train of thought, there are the idea, the algorithm and the method; in the second set for data representation, there are the data structure and the efficiently built processing. The former group instead of the typical iterative solutions deals with recursive design, implementation, testing and inspection [5-6]; the latter deals with recursive type definition, data structures and their constructor and selection operations [7-8].

2. ABOUT PROGRAMMING THEOREMS: ITERATIVE AND RECURSIVE VERSIONS

Let us look at how to work with certain basic scenarios, looking through a certain heap of elements and looking for a single attribute that these elements may or may not have, or just finding the pieces those elements possess, or finding out if those elements possess a certain attribute in two different heaps and relocating them to a new heap could be implemented.

How could these basic scenarios be reduced to a trivial base, and is it more efficient to solve these problems via recursion, or is it inefficient, slow and complicated in modern programming languages?

To find out the answer to this question, we created a small environment, where we can simulate these events, and observe the lifetime of the algorithms. The data we have been using is the Oracle HR scheme database [9]. Our heap of elements consists of employees and departments, those heaps have certain attributes like `name`, `salary`, `id` and so on.

If we are looking for a person in our ship manufacturing company that has the first name '`Bob`' – assuming our company has a finite number of employees – we could just look at all their personal files and check the persons one by one. If the currently active file is the one we are looking for, we take the file, otherwise we get the next file. An algorithm solving this problem could look like this:

```
empList: list of employees
i: integer, default 0;
FindBob()
  while
    i<empList.length AND empList[i].name <> 'Bob'
    i:=i+1
  end while
  Out:
    if i<empList.length then i
      else -1
    end if
end of function
```

This routine solves the problem at once iterating through all the elements. We could willingly reduce the routine to a base case [10-11]:

```
FindBob(int n)
  Out:
    if n<empList.length then
      if EmpList[n].name='Bob' then n
        else FindBob(n+1)
      end if
    else -1
    end if
end of function
```

Let us call this base case solving routine for the first element: `FindBob(0)`.

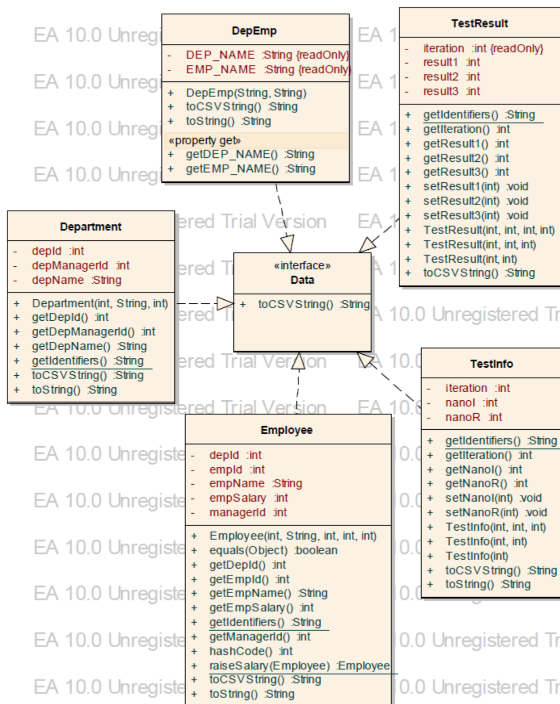

Figure 1.
The Model layer of the application

3. PLANNING

Our plan was to create an environment where we can compare the different approaches with a great emphasis on reusability and reusable classes. So we can later use the same environment for the comparison of different subroutines. We decided to develop the environment in Java 8. For the same reason, we have also been using Model-View-Controller architectural design pattern, so the different layers of the application can be universally replaced by other implementations (Figure 1).

4. IMPLEMENTATION

To compare the efficiency of the two different approaches, we implemented both and measured the run time of the algorithms in a precise manner (in nanoseconds) in Java programming language. Our test service requests a variable argument list of methods to be tested and compared, and assumes that all the data are accessible. The test service creates a complete rundown on the runtime of the tested methods and stores all data on the hard drive; and while doing so it tries not to interfere with the runtime of the tested methods.

To accomplish this, we limited the interval while the passing time itself was being measured.

Step 1: store the current time in nanoseconds in a long variable: `runTime`.

Step 2: invoke the method (from the `vararg` list) via reflection technology.

Step 3: subtract the current time in nanoseconds from `runTime` variable.

The result of the test service is a list of `runTime` variables for each tested method. The service can invoke the methods multiple times, and keep feeding the observed methods with the necessary parameters. The service is running in a background thread, so we can run multiple standalone comparisons simultaneously [12].

We implemented six basic programming theorems via scenarios like the one described before, in two different approaches: both iterative and recursive. These six basic theorems were: sequence count, decision, selection, (linear) search, count and limit selection. And we also implemented six complex theorems in a similar method. These six complex theorems were: copy, select, separation, intersection, union and merge [4, 9, 10].

The model package contains classes required for wrapping the results and the methods that we are testing, some helper classes, and enumerations that are used in the constructors of other classes.

The view package contains the necessary classes for the graphical user interface, the controller package contains the main class of the application that passes the data from the model to the view; this package is responsible for the control of the flow of the application.

Throughout in the program, to ease the burden of further developing the application, we used type generics. To by-pass the problem of storing the methods in a reusable and replaceable form, we used reflection.

5. TESTING

The process of testing and evaluation of the results were made simpler by the graphical user interface. The tabs (Figure 2) show the different tables of the database, and allow switching between different methods that can be tested. At the bottom part of the application, we can see the average of the results of the tested methods' runtimes. We added a nice-looking fancy `javafx.scene.chart.LineChart` to demonstrate the results of the measurement. We also have options to randomize the parameters to be fed to the tested methods.

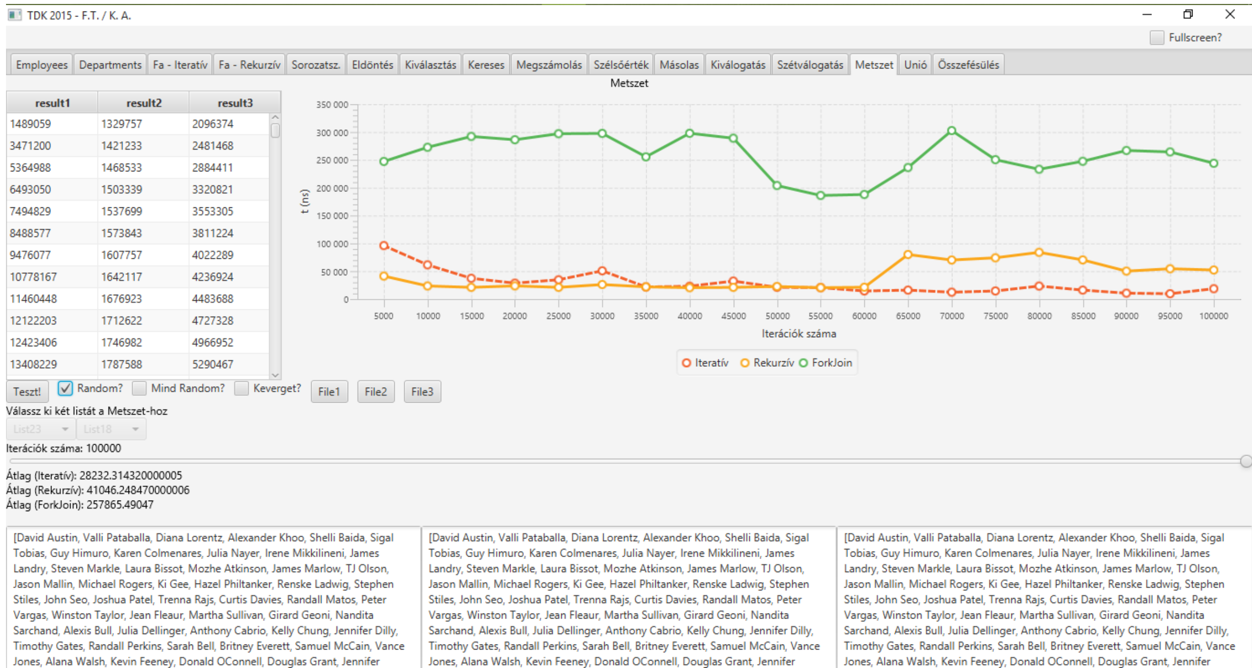

Figure 2.
Showing test results in the application

Programing theorem	Iterative (ns)	Recursion (ns)	ForkJoin (ns)
Sequence count	749	788	
Decision	381	331	
Select (1)	434	336	
(Linear) search	391	372	
Count	978	1144	
Limit selection	1126	1321	
Copy	17469	11990	
Select (more)	4568	2915	
Sepearation	4690	3051	
Intersection	9083	34888	92071
Union	17487	14311	
Merge	13703	9517	

Figure 3.
Comparison of the test results

After testing each programming theorem we summarized the final results and it turned out that the recursive solution was faster in seven occurrences out of the twelve, while the iterative was faster only on five occasions. There was a programming theorem which foregone another one only by a little, for example during selection or making decision.

There were also programming theorems outperforming an implementation compared to another one.

6. RESULTS

On Figure 3 we can see the average of 1,000,000 runs per theorem. Green background marks the implementation that runs faster. As we can see, for the first 6 basic theorems the iterative implementations ran faster (4 times out of 6), while for the last 6 complex theorems the recursive implementations ran faster (5 times out of 6). Summarized for the 12 theorems, the recursive implementations were faster 7 times out of 12.

It is worth noting that among the elementary programming items, the iterative programming was faster by 4-2 ratio; while amongst complex items, recursive became faster by 5-1 ratio. According to the data, the more complex an algorithm, the faster the recursive implementation executes compared to the iterative. This also sums up the assumption that a “critical mass” should be achieved to make it “worthwhile” using the recursive method.

7. FORK/JOIN

In cases where recursion seemed to be less efficient than iteration, we tried to enhance performance through parallelization. We used Java’s built in fork/join framework [13]. The class we used is `java.util.concurrent.RecursiveTask<V>`.

Our `RecursiveTask` implementation is rather simple, it slices one of the lists into two different sublists if the list’s size is greater than a certain threshold, and forks; otherwise it creates a *(n inter)*section of the two lists in a recursive method.

We compared runtime list of this `RecursiveTask` implementation’s `runTime` list with the iterative version of the section creation.

One way to implement the iterative version in Java with the use of `ObservableLists`, `Collections` and such to help decrease the complexity of the code itself looks like this:

```
public ObservableList sectionIterative(
    ObservableList list1, ObservableList list2) {
  Set temp = new HashSet<>();
  for (int i = 0; i < list1.size(); i++) {
    int j = 0;
    while (j<list2.size() && list1.get(i) != list2.get(j))
      j++;
    if (j<list2.size())
      temp.add(list1.get(i));
  }
  return FXCollections.observableArrayList(temp);
}
```

Implementing another solution to the same problem in a recursive method resulted in the following code:

```
private ObservableList<Employee> sectionRecursive(
    ObservableList<Employee> list1, ObservableList<Employee> list2,
    final int n, Set<Employee> temp) {
  if (n>=list1.size())
     return FXCollections.observableArrayList(temp);
  if (list2.contains(list1.get(n)))
    temp.add(list1.get(n));
  return sectionRecursive(list1,list2,n+1,temp);
}
```

After this, we created the specialized `ForkJoin` class for the section creation. The `compute()` method checks if the first list is bigger than a `THRESHOLD`; if it is bigger it forks on the left half of the list, computes the right half of it (containing exactly the threshold number of elements) through the `sectionRecursive()` method, adds the results to a temp set and extends it with the result of `join()` of the second sublist's `RecursiveTask` object.

```
@Override
protected ObservableList<Employee> compute() {
  if (bigger.size()>THRESHOLD) {
    MetszetFJ right = new MetszetFJ(
      FXCollections.observableArrayList(
        bigger.subList(bigger.size()/2, bigger.size())),
        FXCollections.observableArrayList(smaller), false);
    right.fork();
    MetszetFJ left = new MetszetFJ(
      FXCollections.observableArrayList(
        bigger.subList(0, bigger.size()/2)),
        FXCollections.observableArrayList(smaller), false);
    Set out = new HashSet(left.compute());
    out.addAll(right.join());
    return FXCollections.observableArrayList(out);
  }
  else
    return FXCollections.observableArrayList(
      intersect(bigger,smaller));
}
```

However, this parallel approach was without success, it was running even slower than the previous recursive version. Our assumption is that either the problem was not complex enough, or the database we used was too small, so the overhead caused by the memory consuming objects created for the multiple parallel threads was not worth it.

8. SUMMARY

While our work did not result in a straightforward answer to the original question; it, however, does create the assumption that for only a slight, or no sacrifice at the performance, we can solve the same problem with a different, tighter code because recursive algorithms are more often less prolonged than their non-recursive counterparts.

This was sort of a surprise for some of us, we initially assumed that recursion would always be slower than iteration as it would create a high number of unnecessary objects or references to objects on the heap, but it certainly does seem like in some cases, recursion may be a viable option to tighten and tidy up the code a bit, and in certain special cases, may even increase performance a bit.

9. REFERENCES

[1] Szlávi P., Zsakó L.: Módszeres programozás: Rekurzió, Mikrológia 4., ELTE TTK Informatikai Tanszékcsoport, 4. bővített kiadás, 1997
[2] Angster E.: Programozás tankönyv II., 4KÖR Bt., 5., javított kiadás, 1999, ISBN 963 450 957 6 II.K.
[3] Angster E.: Objektumorientált tervezés és programozás, Java, 2. kötet, 4KÖR Bt., 2., átdolgozott kiadás, 2004, ISBN 963 216 513 6
[4] Kaczur S.: Programozási alapok, 1. kiadás, 2009, ISBN 978-963-06-8122-3
[5] Szlávi P., Zsakó L.: Módszeres programozás: Programozási tételek, Mikrológia 19., ELTE TTK Informatikai Tanszékcsoport, 4. javított kiadás, 1999
[6] Recursive factorial | Recursive Algorithms | Khan Academy, https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/recursive-factorial, 2015.10.30.
[7] Iványi A.: Informatikai algoritmusok I., ELTE Eötvös Kiadó, Budapest, 2004, ISBN 963 463 664 0
[8] Pap Gné., Szlávi P., Zsakó L.: Módszeres programozás: Rekurzív típusok, Mikrológia 27., ELTE TTK Informatikai Tanszékcsoport, 3. javított kiadás, 1998
[9] Oracle HR schema, https://docs.oracle.com/cd/B19306_01/server.102/b14198/graphics/ comsc002.gif, 2015.10.02.
[10] Pintér L.: Programozási tételek rekurzív megvalósítása, Mikrológia 10., ELTE TTK Informatikai Tanszékcsoport, 4. kiadás, 1995
[11] Kaczur S.: A rekurzió tanításához használható mintaprogramok Java nyelven, A Dunaújvárosi Főiskola Közleményei, Dunaújváros, XXXI., 2009, ISSN 1586-8567, p. 1-6 (magyar nyelvű szakcikk)
[12] Defining Multithreading Terms, http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html, 2015.10.18.
[13] J. Ponge: Fork and Join: Java Can Excel at Painless Parallel Programming Too!, http://www.oracle.com/technetwork/articles/java/fork-join-422606.html, 2015.10.18.