Algorithms & Data Structures
Dr Michele Sevegnani

Joao Almeida-Domingues[*]

University of Glasgow

January 14th, 2020 – March 25th, 2020

Contents

These lecture notes were collated by me from a mixture of sources , the two main sources being the lecture notes provided by the lecturer and the content presented in-lecture. All other referenced material (if used) can be found in the *Bibliography* and *References* sections.

The primary goal of these notes is to function as a succinct but comprehensive revision aid, hence if you came by them via a search engine , please note that they're not intended to be a reflection of the quality of the materials referenced or the content lectured.

Lastly, with regards to formatting, the pdf doc was typeset in LaTeX, using a modified version of Stefano Maggiolo's class

[*]2334590D@student.gla.ac.uk

1 Recursive Algorithms

recursion traces , linear recursion , binary recursion , tail recursion , recursion trees , incremental algorithms , divide-and-conquer algorithms , merge-sort

1.1 definition. Recursive Function is a function which refers to itself in its definition

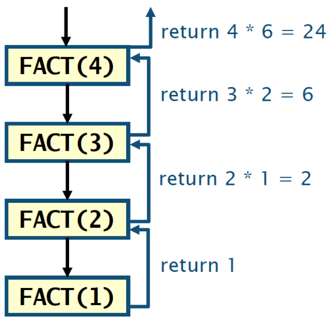
A classical example of a recursive faction is the factorial function, which calls itself always with an argument decreased by one in each call, until the *base case* 1 is reached.

All recursive definitions share these 2 properties, i.e they all have a base case which tells us when to stop calling, and they must reduce the size of the data set with each call

1.1 *Recursion Trace*

A recursion trace allows us to visually trace the execution of recursive algorithms. We draw a box for each call with an arrow pointing downwards connected to the next call, and an arrow pointing upwards with the returning value of that call

1.2 example.



1.2 *Linear Recursion*

1.3 definition. Linear Recursion at most one recursive call with each iteration

1.4 remark. The space required to keep track of the recursive stack grows linearly with n

1.5 remark. Useful when we view a problem in terms of a first/last element plus the remaining set with the same structure (e.g sum array in haskell : `sum(l) = head(l) + sum(tail(l))`)

1.3 *Tail Recursion*

Though useful for designing short and elegant algorithms, recursive can have a high memory cost, since the state of each recursive call must be stored prior to being returned. This costs can sometimes be diminished by using a recursive operation and the very last of the algorithm

1.6 definition. Tail Recursion is linear and the recursive call is its very last operation

In tail recursive implementations, some variable is modified with each call. There's no need to wait until the last call, since the updated variable is immediately passed to the next recursive call. Hence, the state is passed onto the call chain instead of being saved in memory

Recursive ↔ Iterative

A common idiom is converting to/from tail recursive to iterative algorithms. This is achieved by iterating through recursive calls, rather than calling them explicitily
1.7 example. [3] **Recursive**

```
function recsum(x) {
    if (x === 1) {
        return x;
    } else {
        return x + recsum(x - 1);
    }
}

recsum(5)
5 + recsum(4)
5 + (4 + recsum(3))
5 + (4 + (3 + recsum(2)))
5 + (4 + (3 + (2 + recsum(1))))
5 + (4 + (3 + (2 + 1)))
15
```

Tail Recursive

```
function tailrecsum(x, running_total = 0) {
    if (x === 0) {
        return running_total;
    } else {
        return tailrecsum(x - 1, running_total + x);
    }
}

tailrecsum(5, 0)
tailrecsum(4, 5)
tailrecsum(3, 9)
tailrecsum(2, 12)
tailrecsum(1, 14)
```

Iterative

```
function tailrecsum(x, running_total = 0):
    while x >= 0:
        running_total += x
        x -= 1
    return running_total
```

1.4 *Binary Recursion*

1.8 definition. Binary Recursion when an algorithm makes two recursive calls

1.9 example. Fibonacci : FIB(n-1) + FIB(n-2)

1.10 remark. Highly inefficient. Note how the same computations are made several times over, in different calls. In the recursion tree there are $O(2^n)$ recursive calls, i.e exponential complexity

recall memoization technique from 1P

2 Algorithm Design Paradigms

2.1 *Incremental*

The solution of a problem is built one element at a time

2.1 example.
Sort sub array ; pick unvisited node ; add to sorted subarray

```
INSERTION-SORT(A)
    for j = 1 to n-1
        key := A[j]
        i := j-1
    while i >= 0 and A[i] > key
        A[i+1] := A[i]
        i := i-1
        A[i+1] := key
```

2.2 *Divide-and-Conquer*

Recursive algorithm, with 3 steps

1. Divide into smaller subproblems equivalent to the original
2. Conquer by solving subproblems recursively
3. Combine the solutions to the subproblems to give the general solution

2.2 example.

Merge-Sort (1) split the array into two subarrays of size $\frac{n}{2}$; (2) scan two smaller arrays simultaneously ; (3) add minimum to sorted array, incrementing the pointer of the array from where the value was removed; when fully scanned copy the remaining sub array to main array

References

CS2 Software Design & Data Structures **calculating˙program˙running˙time**

CS2 Software Design & Data Structures. URL: `https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/AnalProgram.html`.

Goodrich et al.: Data Structures and Algorithms in Java, 6th Edition **goodrich˙tamassia˙2014**

Michael Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java, 6th Edition*. John Wiley & Sons, 2014.

Hochstein: What is tail recursion? **hochsteinSO**

Lorin HochsteinLorin Hochstein. *What is tail recursion?* Aug. 1958. URL: `https://stackoverflow.com/questions/33923/what-is-tail-recursion`.