COMP6700/2140 Recursive Operations

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

May 2017

Topics

1. Recursive Functions
2. Tail Recursion
   - Recursion stack
   - JVM Stack Machine Architecture ☹
3. Tail Recursion Optimisation
4. Recursions vs Iterations
5. Best Algorithm

Recursive methods

Methods which call themselves

A basic problem of calculating the *factorial of N* (trite, "cheesy" example — sorry):

$$N! = N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot 2 \cdot 1 = N \cdot (N-1)!$$

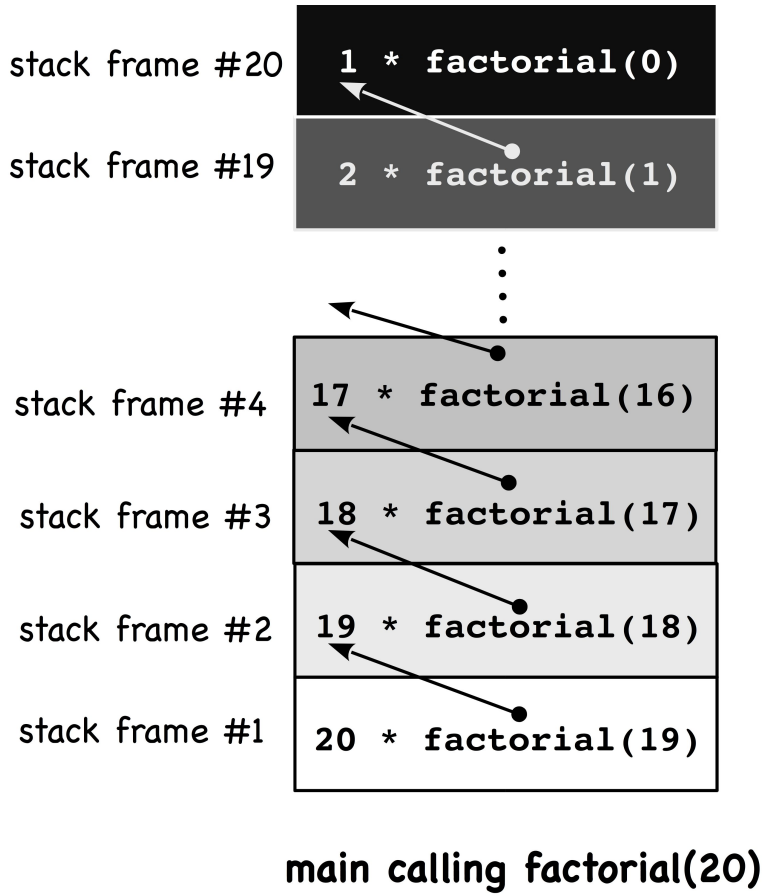
One implementation is "old fashioned" iterative:

```
public static int factorial(int n) {
   int r = 1;
   for (int i=1; i<=n; i++)
      r=r*i;
   return r;
}
```

The recursive implementation:

```
public static int factorial(int n) {
   assert (n >= 0) : "The factorial is tricky if the argument is negative!"
   if (n == 0) return 1;          // Base Case: recursion stops
   return n * factorial(n-1);     // Generic Case: a recursion call is made
}
```

Recursion call stack



Tail recursion

What if the body of a value returning recursive method can be written in such a way that the only recursive call is the one which is made in the `return` statement? For `factorial`, this would require *almost* pointless and trivial modification:

```
public static int tailFactorial(int n, long a) {
   assert (n >= 0) : "The factorial is tricky if the argument is negative!"
   if (n == 0) return a;
   return tailFactorial(n-1,n*a);
}
```

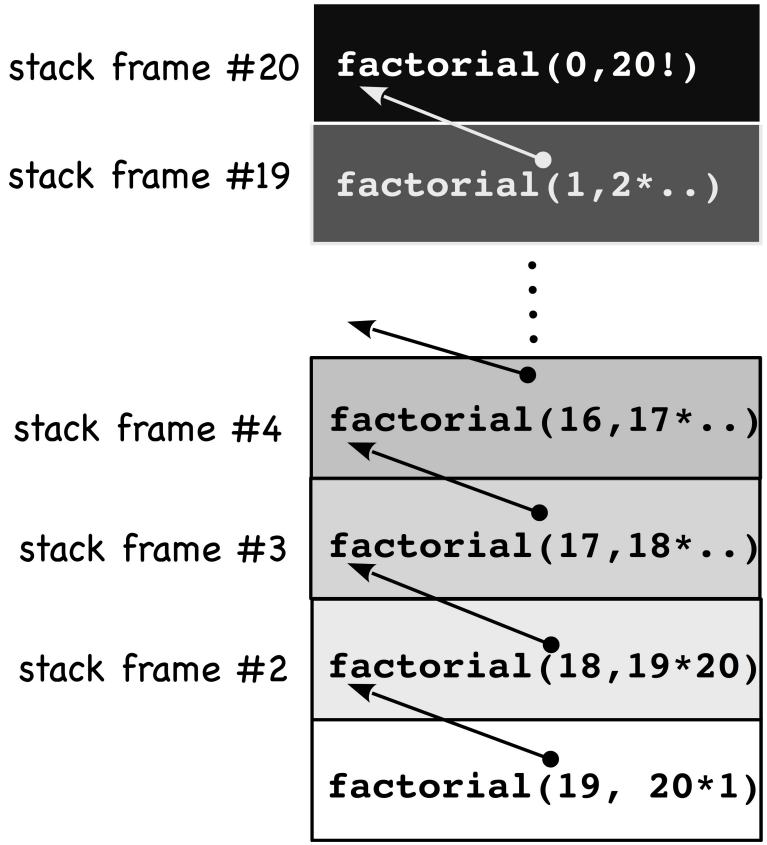
It is obvious, that mathematically `factorial(x)` and `tailFactorial(x,1)` are the same function. Yet in computing:

> *It is often easier to do more work rather than less.*

(quoted from the book *From Mathematics to Generic Programming* by Alexander A. Stepanov and Daniel E. Rose)

So, what's the point?

Tail recursion call stack



Tail Recursion Optimisation

Let's rewrite again, but using reassignment of the passed arguments:

```
public static int tailFactorial(int n, long a) {
   if (n == 0) return a;
   a *= n; n--;
   return tailFactorial(n,a); // when called, the above code is repeated verbatim
}
```

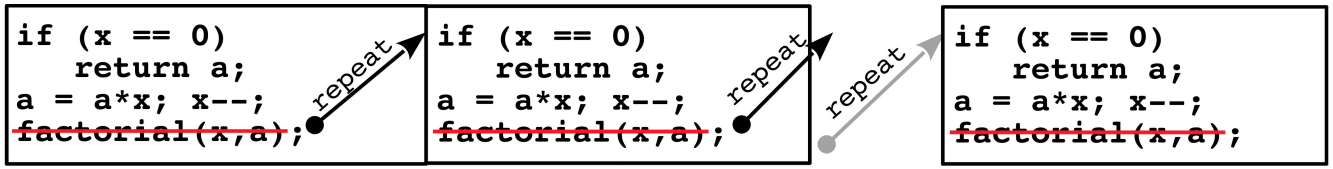
Such form of a tail recursion — when all the tail recursive calls are made with formal arguments of the method being the corresponding arguments — is called a *strict tail recursion*. The recursive call in the return statement results in all preceding statements of the method body repeating themselves. Nothing gets changed if the last return statement is dropped, and instead the remaining body is repeated until the base case (`n = 0`) is realised and the iteration terminated:

```
public static int tailFactorial(int n, long a) {
   while (true) {
      if (n == 0) return a;
      a *= n; n--;
   }
}
```

The recursion has been transformed into a loop — **Tail-Call Optimisation!**

Tail recursion optimised

- Recursion call stack is gone — no heavy (vertical) burden to carry.
- Computation is horizontal now, the stack isn't consumed, there is no bound on how long the repetition can go.



OR

```
while (true) {
    if (x == 0)
        return a;
    a = a*x; x--;
}
```

Languages like *Lisp* could perform the tail recursion optimisation automatically, as the part of their code compilation/interpretation. *Java* (and hence other JVM languages) **cannot** perform TCO due to restrictions imposed by the JVM architecture. But there are means to deal with this problem (wait till F6 lecture on "Recursion, Corecursion and Memoization in Functional Java Programming").

The best algorithm

Imagine, 10% improvement in performance of the search engine in online mega-businesses like *Google* or *Amazon*. The difference is tens of millions of dollars. A 10% more efficient algorithm in a mobile battery-powered device means a few hours of longer work. A 10% less power consuming servers may translate into 1–2% less gross country's energy consumption (in South Korea, about 7-9% of total power is spent on running the internet and telephony servers).

What affects the algorithms performance?

1. Mathematical efficiency ($\mathcal{O}, \Omega$ and Θ);
2. Time and Space resource requirements — CPU cycles *T(N)* and memory *P(N)* — and the cost of their provision (how fast data can be brought to the processing unit, what is the frequency of cache misses *etc*);
3. Which data structures are used and how do they map into the processor architecture (size and layout of cache, bus parameters etc); as fas as the role of cache is concerned, the situation can be similar to the binary search bug which reveals itself when we move to larger data sizes. Here, instead of becoming buggy, an algorithm can drastically worsen its performance — an example is the *HeapSort*, which is on par with *QuickSort* ($\mathcal{O}(N \log N)$) and doesn't have the "worst case" problem, like QuickSort ($\mathcal{O}(N^2)$ on bad inputs). This changes when the size becomes large, and the *cache misses* begin affect the performance.

With all these factors playing an important role, the finding and choice of the best algorithm has been and still is a very non-trivial and important problem. Finding and implementing algorithms will remain an important problem the IT industry has always to deal with.

Recursions vs Loops

When one is better than another

- Iterations are most often faster (recursions have *method stack* overhead), but this advantage is not crucial
- Iterations require less memory (recursions have limited depth due to finite stack size), and this maybe crucial
- Recursions often (not always!) result in a simpler (often, *much* simpler) implementation due to the algorithm recursive nature (either due to the structure of mathematical definition, like *factorial* or *Fibonacci sequence*, or due to recursive nature of the data structure on which the algorithm operates, *eg*, trees)
- Non-recursive algorithms on recursive data structures are possible, *eg*, search in a binary tree (but it also involves a stack)
- Recursive algorithms on non-recursive data structures (arrays, lists ...) are possible (*Quicksort* and others)
- Proof of correctness for recursive algorithms often greatly assisted by the *mathematical induction* method
- When using recursion, always provide the base case (cases), and ensure that the recursive calls converge to one of the base cases. Remember of the stack size limitation! Never use recursion for processing user inputs!

Other algorithms

Study of algorithm is at the core of Computer Science. *"Simple fundamental algorithms is the ultimate portable software"* (R. Sedgewick, J. Bentley). *"Not so fast, guys!"* (Alexander Stepanov)

Algorithm Intensive CS Fields

1. Sorting and searching on various data structures
2. Graphs and Networks (include data mining)
3. Optimisation (incl. parse trees in compiling)
4. Number theory including encryption (factorisation, elliptic curve arithmetic)
5. Computational geometry and computer graphics
6. Numerical computations (matrix multiplication and spectral analysis, Newton's method etc)

Few references

- "Introduction to Algorithms", by Thomas H. Cormen, Sharles E. Leiserson, Ronald L. Rivest and Clifford Stein, McGraw-Hill, 3d Ed, 2007
- "Fundamentals of Algorithms", by Gilles Brassard and Paul Bratley, Prentice-Hall, 1996
- "The Art of Computer Programming" 3rd Edition, by Donald Knuth, Addison-Wesley, 1997–1998; volumes:
  1. "Fundamental Algorithms"
  2. "Seminumerical Algorithms"
  3. "Sorting and Searching"

Where to look for this topic in the textbook?

Very little about recursive algorithms and algorithms in general is discussed in the textbook (it teaches *users* how to program, not *developers* of programming resources, like libraries and API). You should seek other sources of knowledge in these areas, both in printed form and (much more and growing) online.

- Hortsmann's Core Java for the Impatient, Ch. 3.3, 3.8
- Oracle's Java Tutorial chapter Algorithms — again, it deals with the JFC usage, not much with the intricacies of generic algorithm implementation.
- One modern (and relatively light) text is "Algorithms in a Nutshell", 2ed, by George T. Heineman, Gary Pollice, and Stanley Selkow (O'Reilly, 2017)