Summary of module MA1

Recursion

- Divide the problem into (one or more) subproblems of the same type but smaller.
- Solve the subproblems.
- Combine the solutions of the subproblems into a solution of the original problem.

There must be one or more base cases.

Using the simplest base cases (eg 0 instead of 1 in the Tower of Hanoi, or an empty list instead of a list with one element) usually leads to the simplest code.

Why recursion?

- General technique for problem solving.
- Easy to find the solution.
- Easy to find efficient solutions.
- Natural in many contexts.
- Especially good for recursively defined structures.

But

- Easy to produce hopelessly slow programs.
- Be aware problems with the recursion depth.

What is decisive?

Decisive are the number of subproblems and the size of the subproblems.

- If we have two or more subproblems of *almost the same size* as the original problem, we have exponential growth.
  Example: Tower of Hanoi, the Fibonacci numbers.
- Usually better with *two* subproblems that are *half the size* than *one* subproblem that is *almost as big* as the original problem
  Example: merge sort – insertion sort
- As a rule, it is good to *balance* the size of the subproblems.
- Avoid recursion over long structures - stack depth issues.

Asymptotic notation

A way of describing how the time of an algorithm grows with the problem size independent of computer, programming language, etc.

When do we use $\mathcal{O}, \Theta,$ and Ω respectively?

- Θ gives most information.
- $\mathcal{O}$ is an *upper* limit (not necessarily tight).
- Ω is a *lower* limit.

Big O, Omega or Theta?

- If you have a "good" $\mathcal{O}$ function, it can be used to say that an algorithm is good.
  Example: If you have invented a sorting algorithm, it is good to be able to say that it is $\mathcal{O}(n \log n)$ but meaningless to say that it is $\mathcal{O}(n$^2).

- If you want to say that an algorithm is bad, you can use Ω.
  Example: If someone comes up with a sorting algorithm, you can say it's not very good if it's $\Omega(n$^2) and it's lousy if it's $\Omega(n$^3).
  However, it is meaningless to say that it is $\Omega(n \log n)$.

- Θ is most informative. Use if possible.

What is good and what is bad?

- $\Theta(a^n)$ is *bad* if $a > 1$.
- $\Theta(\log n)$ is *much better* than $\Theta(n)$.
- $\Theta(n \log n)$ is *much better* than $\Theta(n^2)$.
- $\Theta(n)$ is *not much better* than $\Theta(n \log n)$.

Time estimates

If we know that an algorithm is $\Theta(f(n))$ then we can estimate the time taken $t(n)$ for large problems with the expression

$$t(n) = c \cdot f(n)$$

and estimate the constant by measuring the time for some n.

You should verify the model by measuring the time for different values of n.

Do not use too small n – the larger the values you measure for, the more accurate the estimates will be.

Examples

To verify the complexity of a particular algorithm, it is good to have a strategy for how n should be varied.

Doubling the value is good if you have (think you have) polynomial complexity :

- For a $\Theta(n)$ algorithm then the time should be doubled.
- For a $\Theta(n^2)$ algorithm: $\frac{t(2n)}{t(n)} = \frac{c \cdot (2n)^2}{c \cdot n^2} = 4$
- För en $\Theta(n^3)$ algorithm: $\frac{t(2n)}{t(n)} = \frac{c \cdot (2n)^3}{c \cdot n^3} = 8$

Examples

- For a $\Theta(\log n)$ algorithm it is good to square: $\frac{\log n^2}{\log n} = 2$
- For a $\Theta(n \log n)$ algorithm dubling is useful:

$$\frac{t(2n)}{t(n)} = \frac{c \cdot 2n \log 2n}{c \cdot n \log n} = 2 \cdot \frac{\log 2 + \log n}{\log n} = 2 + \frac{1}{\log n} \approx 2 \text{ for large } n.$$

- For exponential growth, ie $\Theta(a^n)$ is is good to use n och $n+1$:

$$\frac{c \cdot a^{n+1}}{c \cdot a^n} = a$$

Should we care?

Yes! There are still many problems where computing power limits us.

A selection :

- physics och technology: aerordynamics, weather forecasts, …
- biology: bioinformatics, genome sequencing, …
- real-time systems: robots, self-driving cars, …
- animation: computer games, film industry, …
- information search : google, …
- artificial intelligence, machine learning

But

Citation from ”*The elements of programming style*” by Kernighan and Ritchie:

- Correctness is much more important than speed!
- Do not sacrifice clarity for small efficiency gains!
- Do not sacrifice simplicity for small efficiency gains!
- Do not sacrifice modifiability for small efficiency gains!
- If the program is too slow: Find a better algorithm!

Some Python-details

You can declare functions inside functions.

First example:

```
def power(x, n):
    def sqr(x):
        return x*x

    if n<0:
        return 1./power(x, -n)
    elif n==0:
        return 1
    elif n%2==0:
        return sqr(power(x, n//2))
    else:
        return x*sqr(power(x, (n-1)//2))
```

Some Python-details

Second example:

```
def fib(n):
    memory = {0:0, 1:1}

    def _fib(n):
        if n not in memory:
            memory[n] = _fib(n-1) + _fib(n-2)
        return memory[n]

    return _fib(n)
```

Some Python-details

Functions are *objects* that can be stored in variables, lists, dictionaries, …

```
sort_functions = [ins_sort_iter, merge_sort, psort, sorted]

for sort in sort_functions[1:]:
    print(f'\n ***{sort.__name__}***')
    for n in [100000, 200000, 400000, 800000]:
        lst = []
        for i in range(n):
            lst.append(random.random())
        tstart = time.perf_counter()
        lst = sort(lst)
        tstop = time.perf_counter()
        print(f" Time for {n}\t : {tstop - tstart:4.2f}")
```

The end