F2

Reading reference: chapter 2 + slides

- Algorithm complexity
- Big O and big Ω
- To calculate running time
- Analysis of recursive Algorithms

Next time:

Litterature: Chapter 4.5-4.7, slides

- Greedy algorithms
- Minimum spanning trees
- Minimum spanning tree property
  (called cut-property in book)
- Kruskals algorithm
- Merge-Find sets (MF-sets)
  (also called Union-Find sets)
- (Priority queues)
- Clustering

How do we evaluate algorithms?

What does it mean to evaluate an algorithm?

Is it: “Does it work”?

Is it: to be able to compare two algorithms?

Is it: optimal (in speed, memory etc.)

Is it: How to make i better?

3 levels

1. It must be correct - it must work.

2. Must be easy to understand, code, maintain etc.

3. Should use computer resources well (speed, memory,...) e.g. should be “efficient”

The result of the evaluation of computer resources is called the algorithm's complexity.

Computing science “folklore”

- *The fastest algorithm can frequently be replaced by one that is almost as fast and much easier to understand.*

- Make it work before you make it fast -

If a program doesn’t work it doesn’t matter how fast it runs.

Algorithm complexity

We will now look at how to mathematically analyse algorithms to determine their resource demands.

Definition:

The complexity of an algorithm is the "cost" to use the algorithm to solve a problem.

The cost can be measured in terms of

- executed instructions (the amount of work the algorithm does),
- running time,
- memory consumption
- or something similar.

So complexity is a measure of the algorithms resource demands, it's not about how hard the algorithm is to understand

The most interesting resource seems to be running time but is that a good measure?

The running time depends on many things:

1. The programmers skill and
   the programming language.
2. The computers machine language + hardw.
3. The code that the compiler generates.
4. **The size of the problem**
   (e.g. nbr of inputs).
5. **The nature of input** (sorted/unsorted).
6. **Time complexity of the algorithm itself**,
   of the method chosen.

Items 1-6 is the base for “empirical” complexity, we can “measure” the time the algorithm takes on some input - the running time.

Drawbacks with empirical complexity:

- experiments can't cover all inputs
- you must implement the algorithm
- different algorithms has to be run on the same software/hardware/programmer

-

Items 1-3 are (usually) not affected by the algorithm used and will only affect running time with a constant factor that may vary with different languages, computers etc.

We like to ignore these constant factors and create a theoretical analytical framework...

Item 4-6 represents the theoretical complexity where we evaluate the algorithm “itself”.

+ see drawbacks with empirical complexity!

Drawbacks with a theoretical framework:

- constants can matter
- software/hardware/programmers can give you subtle problems for instance Strings in Java that is not “visible” here

Elementary operations

We will measure the amount of work the algorithm does in terms of “elementary operations”.

Elementary operations are assumed to require one “cost”- unit e.g. it’s an operation whose complexity can be bounded by a constant.

Complexity is a relative concept, only interesting together with the corresponding elementary operation.

Problem	Problem size	Elementary op.
Find x in list	Number of elements in list	Comparing x to list element
Multiplication of two matrices	Dimension of matrices	Multiplication of two numbers
Sort an array	Number of elements	Comparing array elements or movement
Traverse a tree/graph	Number of nodes	Follow a pointer

Problem size influence complexity - Logarithmic cost criteria

Since complexity depends on the size of the problem, we define complexity to be a function of problem size

Definition:

Let T(n) denote the complexity for an algorithm that is applied to a problem of size n.

The size (n in T(n)) of a problem instance (I) is the number of (binary) bits used to represent the instance.

So problem size is the length of the binary description of the instance.

This is called **Logarithmic cost criteria.**

Ex: input $x_1, x_2, x_3, \ldots, x_k$

size = $\log(x_1) + \log(x_2) + \ldots + \log(x_k)$

Uniform cost criteria:

Iff you assume that

- every computer instr. takes one time unit,
- every register is one storage unit
- and that a number always fits in a register

then you can use **the number of inputs** as problem size since the length of input (in bits) will be a constant times the number of inputs.

Example:

bigProblemToSolve(x_1, x_2, x_3, …, x_n) {...}

Inputs are x_1, x_2, x_3, …, x_n

Then the size of the problem is

$\log(x_1) + \log(x_2) + \log(x_3) + \ldots + \log(x_n)$

this is $\leq n*\log(\max(x_i))$

and $\log(\max(x_i))$ is a constant $\leq \log(\text{int.max})$

so $\leq n * k$

This is called Uniform cost criteria.

Asymptotic order of growth rate and Big O

Small n is not interesting!

The growth rate of complexity e.g. what happens when n is big, or when we double or triple the size of input, that's interesting.

We call this asymptotic growth rate and we need a way to express this mathematically:

"**Big O**". (Ordo of f, "big O" of f, order of f)

Let, for a given function f, O(f) be the set of all functions t that are asymptotically bounded above by a constant multiple of f.

Formally:

$$O(f(n)) = \{t \mid (\exists c > 0)(\exists n_0 \geqslant 0)(\forall n \geqslant n_0)(t(n) \leqslant c * f(n))\}$$

$$\text{where } f, t : N \rightarrow R^{0+}, c \in R^{+}, n_0 \in N$$

or in words:

T(n) is O(f(n)) if there exists constants $c>0$ and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $T(n) \leq c^* f(n)$.

We write $T(n) \in O(f(n))$.

“Rules” of ordo notation

$O(f)+O(g) = O(f + g) = O(\max(f, g))$

$c * O(f) = O(c * f) = O(f)$

$f * O(g) = O(f) * O(g) = O(f * g)$

$O(O(f)) = O(f)$

Big Ω (Omega)

Ordo notation gives us a way of talking about upper bounds

“it takes no more time than...”

We also like to be able to talk about lower bounds, to be able to say that

“it takes **at least** this much time to...”

Let, for a given function f, **Ω(f)** be the set of all functions t that are asymptotically **bounded below** by a constant multiple of f.

$$\Omega(f(n)) = \left\{ t \,\middle|\, (\exists c > 0)(\exists n_0 \geqslant 0)(\forall\, n \geqslant n_0)(t(n) \geqslant c * f(n)) \right\}$$

$$\uparrow$$

Nature of input influence complexity

Complexity also depends on the nature of the input.

(for instance: sorting n numbers can be much quicker if they are (almost) sorted already)

Therefore we use the worst case as a measure of complexity

Definition:

Let T(n) be the maximum cost over all n to apply the algorithm on a problem of size n.

This gives us an upper bound of the amount of work the algorithm performs, it can never take more time.

When we say that an algorithm takes time $\in O(n^2)$ we mean that it takes **no more time than** a constant times n^2, for large enough n, to solve the problem in worst case.

When we say that an algorithm takes time $\in \Omega(n^2)$ we mean that it takes **at least** a constant times n^2, for large enough n, to solve the problem in worst case.

In both cases this does not mean that it always will take that much time, only that there are at least one instance for which it does.

Example:

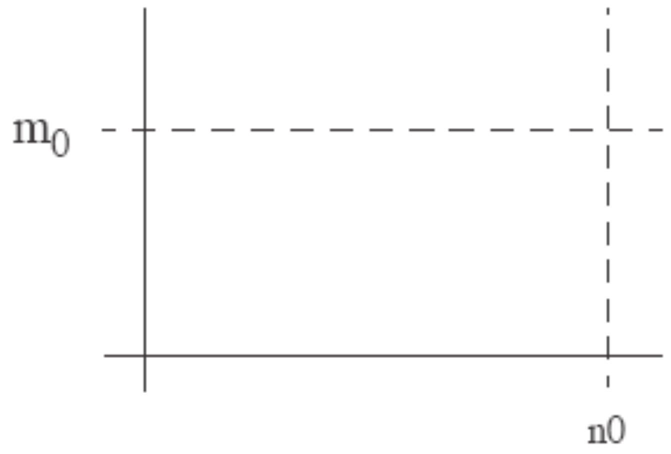
Looking for x in a unsorted list with n elements takes $O(n)$ and $\Omega(n)$ time in the worst case but if x is first in the list it takes $O(1)$ time.

Functions of more than one variable

What if we have several parameters T(n, m)?

(for instance matrix indices as parameters)

We can generalize:



$$O(f(n,m)) = \left\{ \begin{array}{c} t \mid (\exists c > 0)(\exists n_0, m_0 \geq 0)(\forall n \geq n_0)(\forall m \geq m_0) \\ [t(n,m) \leq c \times f(n,m)] \end{array} \right\}$$

But more often: try to express complexity as a one parameter function for instance by adding or multiplying the parameters:

f(n') = f(n*m) or f(n') = f(n+m)

Conditional notation:

O(f(n) | P(n)) where P(n) is a predicate.

$O(n^2 \mid n \text{ odd})$ valid only for odd n

$O(n\log n \mid n=2^k)$ n must be power of 2.

Calculating complexity

1. Only executable statements “costs”.

2. **Simple statements** like assignment, read / write of simple variables and so on take O(k).

3. For a **sequence of statements** the cost is the sum of the individual costs.

4. For **decisions like “if”** and “case” statements you add the cost for the decision (usually constant) plus the cost of the most expensive alternative.

```
   if n<0 then print n;
   else print an array 1..n
```

5. **Loops** with constant step: the sum over all turns of the time to evaluate the condition plus the body. (i.e. the number of turns times the largest time of the body over all turns)....

6. **Multiplicative loops** …. see --->

7. **Subprograms** are analysed according to the rules above. Recursive algorithms are a bit more difficult. We must express T(n) as a recursion equation ....
Solving them can be a bit tricky.

Examples loops

Sum up the numbers 1 to n, elem.op.= aritm. operations, reading a variable is free

pedantic analysis:

```
1  int sum = 0;           (1 op)
2  for ( int i=1;         (1 op)
3            i<=n;        (1 op every turn)
4               i++ ){    (2 op every turn)
5     sum = sum + i;      (2 op every turn)
   }
```

Gives us the complexity function:

```
                      Row
T(n) = 1 +            1
       1 +            2
       (n+1)*1+       3    (nbr of turns)*1
       n*2            4
       n*2            5
```

`     = 3 + 5*n` = an exact solution $\in$ O(n)

mathematically correct estimate:

$$2+\sum_{i=1}^{n} 5=2+5\sum_{i=1}^{n} 1=2+5n\in O(n)$$

rough estimate: n turns $*$ cost of body = n

Example:

```
for i = 0..n loop
   for j = 1..n loop
      something O(1)
   end loop
end loop
```

mathematically correct:

(the sum over all turns of the time to evaluate the condition plus the body)

$$T(n) = \sum_{i=0}^{n} \sum_{j=1}^{n} O(1)$$

(known sums: $\sum_{i=1}^{n} 1 = 1+1+\ldots+1 = n \in O(n)$

$\sum_{i=1}^{n} i = 1+2+\ldots+n = \frac{n(n+1)}{2} \in O(n^2)$)

rough estimate:

i.e. the number of turns times the largest time of the body over all turns

T(n) = $n * n * 1$

or T(n) = $(n+1) * n * 1$

Multiplicative loops

with multiplication/division with a constant like

```
control = 1;
while control <= n loop
   something O(1)
   control = 2 * control;
end loop;
```

After k iterations we have

control = $2^k \Rightarrow k = {}^2\log(\text{control})$

since k is the number of iterations
and control = n at the end of the loop
then k = log(n)

Observe that the base of the logarithm doesn't matter since the logarithm of a number in one base is equal to a constant times the logarithm in another base so $O({}^a\log n) = O({}^b\log n)$.

Usual Ordo functions

Polynomial complexity (or better):

(Polynomial complexity is anything smaller than or equal to a polynom)

O(1), O(k):	constant complexity
$O({}^{y}\log n)$:	logarithmic complexity
O(n):	**linear** complexity
O(nlogn):	lin-log complexity
$O(n^2)$:	quadratic complexity
$O(n^3)$:	cubic complexity
$O(n^4)$:	
$O(n^5)$:	

Worse:

$O(2^n)$:	exponential complexity
O(n!)	factorial complexity

And combinations of these like

$O(n * {}^{2}\log n)$, $O(n^3 * 2^n)$, ...

Recursive functions

The faculty function is defined as

$$n! = \begin{cases} 1 & if\, n=1 \\ n(n-1)! & otherwise \end{cases}$$

```
int fac(int n)
   if n <= 1 then
      return 1
   else
      return n*fac(n-1)
   end if
end fac
```

If the call to fac(n) take T(n), then the call to

fac(n-1) should take T(n-1). So we get

$$T(n) = \begin{cases} c_1 & if\, n=1 \\ T(n-1)+c_2 & n>1 \end{cases}$$

whose solution is O(n).

Mergesort is a famous sorting algorithm

```
array mergesort (array v; int n)
   // v is an array of length n
   if n = 1 then
      return v
   else
      split v in two halfs v1 och v2,
                       with length n/2
      return merge(mergesort(v1, n/2),
                   mergesort(v2, n/2))
   end if
end mergesort
```

Merge takes two sorted arrays and merges these with one another to a sorted list. If T(n) is the wc time and we assume n is a power of 2, we get:

$$T(n) = \begin{cases} c_1 & \text{if } n=1 \\ 2T(n/2)+c_2 n & \text{if } n>1 \end{cases}$$

The first “2” is the number of subsolutions and n/2 is the size of the subsolutions

c_2n is the test to discover that n≥1, (O(1)), to break the list in two parts (O(1)) and to merge them (O(n)). The solution is O(nlogn)

Solving recursion equations

means that we try to express them in closed form, i.e without T(...) term on the right hand side.

- Repeated substitution - expand the identity until all terms on the right hand side only contains T(c) which is a constant.
- Guess a solution f(n) and prove it to be correct, that $T(n) \le f(n)$, with induction. You need experience with guessing!
- Simple transformations of range or domain (for example Z-transform)
- Generating functions (special case of transforming the domain)
- Special methods like
  - Direct summation.
  - Cancellation of terms.
- Methods for solving difference and differential equation
- Tables and books
- Give up...

Strategy for complexity calculations

1) Set up the requirements, principally which elementary operations (EO) you use
i.e. what costs are you counting.
(and possibly what you are NOT counting)

2) **Set up the formula for the complexity**
Do that rather exactly (mathematically correct) and motivate carefully what you do.

Example: in the sum

$$3+\sum_{i=1}^{n} 5$$

you must motivate where 3, i=1, n and 5 came from and why a sum is appropriate, referring to the pseudocode/algorithm.

This can be done by having row numbers in the pseudocode and writing like

row 1-3 does one EO each

row 4 is a loop that start with i=1 since ....

3) **Solve the formula**

Think about what the result is going to be used for:

- If you only want an O(.):

simplify as needed if it makes solving the formula easier. (But don't do it in a routine-like fashion)

And think about what you do, example:
a single loop with a O(n) body costs as much as a double loop with constant body.

- But if you need to compare different algorithms with the same O(.) you need to solve more precisely.

Always motivate what you do. Math can be motivated like this:

"..." is the formula that you are working with

.... = {divide everywhere with 4}

.... = {use formula for geometric sum}

.... = and so on

Trivial things need not be motivated but be over-explicit rather than the opposite.

With long formulas with many subexpressions (like double sums for instance) you can solve the subexpressions separately so in

$$3+\sum_{i=1}^{n}\sum_{i=0}^{n-1}5$$

you can solve the inner sum before or after this calculation and just insert the result here

.... = {solution of inner sum, see below}