COMPLEXITY OF PROGRAMS, ALGORITHMS, AND PROBLEMS

Just as you would chose to walk, bike, use the moped, motorcycle, train, boat or airplane, depending on the appropriate occasion, so would you chose the appropriate program/algorithm for the occasion. A quick-and-dirty or a well thought-out and efficient one. But what criteria do you use and how do you decide?
In order to make choices between two or more alternatives, one would like to compare these choices, quantify them, and map these choices into the real number system or some other system with a total ordering so that one alternative can be shown to be better than another since its mapped value is smaller than the mapped value of other choices. If several criteria can be used, then it is customary to weigh these according to some weighing scheme. First, let us distinghuish between the objects we are trying to compare with each other: programs, algorithms, and problems. These are in increasing level of abstraction.

Complexity of Programs

A program depends on the problem it is designed to help solve, the algorithm and datastructures used in the method to find the solution, but also the particular programming environment like machine used, memory available, language used, etcetera. For comparisons of programs, one can look at one or more of the following criteria:

1 programming time.
2 debugging time.
3 number of statements (static space requirement).
4 number of executable statements.
5 number of executed statements.
6a number of executed statements if you are lucky with the input (best case)
6b number of executed statements if the input is neither good or bad (average case)
6c number of executed statements if you have bad luck with the input (worst case)
7 number of executable operations.
8 number of executed operations. (best-average-worst).
9 number of executable operations of a certain kind or type (arithmetical, logical, etc.). (best-, average-, worst).
10 number of executed supporting operations, (e.g. page-swaps, memory accesses, tape-mounts etc.), (best-average-worst).
11 dynamic space requirement, (best-average-worst), ie space needed when running.
12 maintenance time.
13 coding time.
14 aestetic value.
15 level of sophistication.
16 ease of proving correctness.
17 prestige that comes with a program
18 number of voluntary I/O when running
19 number of involuntary I/O when running (page faults).
20 number of independent modules.

Many more are possible, and many of these are qualified by 'best' case, 'worst case' or 'average case', to signify the cost 'if you are lucky and hit on a particular easy case for this program', 'if you are not particular lucky, but not particularly unfortunate either' (also: average over many instances of the problem). The basic problem with just about all the above criteria is that they are appropriate only for programs, and thus depend on the particular programming language, the particular compiler used, the particular machine used to run it, and even, (in a shared environment) on the particular load on the system when you run/ran the program? It is conceivable that with different languages, compilers, machines, and load the performance measure of the program would be different. (But then again, it would not be the same program, would it?).

Complexity of Algorithms and datastructures

Most of the measures above can not be used for the complexity studies of algorithms and datastructures. More importantly, you would not want to use these measures, because these can often only be measureed after you have decided to implement the algorithm. When comparing algorithms, we want to compare these, but we do not want to implement these if we can help it. Of course, one could try to focus on one particular language, compiler, machine, and load, and dictate that every algorithm be measured against this benchmark environment. Indeed, Church's thesis states: 'Any algorithm can be expressed on a Turing Machine and any Turing Machine expresses an algorithm.'. We could and should therefore focus the attention on Turing Machines, but this will quickly lead to having to carry 'way too much detail' and is rather unrealistic. We want to study both space- and time complexities of algorithms: How much space is needed to store this data structure and how much space and how much time is needed to execute this control structure? Trying to narrow down our choice for the criteria to use in our course, we will try to find some metric that is

- Machine independent (and independent of environmental concerns like Operating System support routines and such)
- Language independent.
- Environment independent (load on the system, ...)
- Emotional independent (i.e. two different persons should come up with the same answer, assuming they both use correct methodology, it eliminates the personal gain criteria).
- Amenable to Mathematical study
- Realistic

This insistence on independence creates however some problems (but avoids many more). The problems it creates are: 1) How can we measure time if we can not use a particular machine? 2) Even if you assume that a certain operation takes t_{op} time, how can we count the number of times that a certain operation is executed if we don't even know whether our eventual language will have this operation as a native operation? The big problem it avoids however has much to do with these seemingly disadvantages: It allows you to study the complexity of an algorithm before you go through the trouble of implementing, coding, testing, etc. only to find out that the program needs more space than you have or more time than you are willing to wait for it.

The actual counting is not very hard if you follow the basic rules for some of the fundamental control-structures like statement, compound statement, for-loop, while-loop, if-then-else, case, ... But you end up with an expression that is so complicated that it is hard to realize what it actually means. If you have two different algorithms, how do you decide which one is better? Is one algorithm always better? Does it depend on the size of the input? If so, we need a system that allows us to compare two functions over the range of allowable input sizes, but at the same time allows us to look only at the 'most important part' of the timing function. Furthermore, we have to decide on which domain to compare the two (or more) functions, as it might very well be that these functions cross over somewhere. It is customary to investigate these timing functions (and space functions) at the region for arbitrary large input sizes. Enter the world of asymptotics and its own particular language. (In particular, see another handout if you are not familiar with O-notation). This region may not apply in your particular situation where you need to sort say only 3 million records, but will give you an indication as to what you can expect from certain algorithms.

When we study the complexity of an algorithm, we better first define what an algorithm is. Again, according to Church: 'Any algorithm can be expressed on a Turing Machine and any Turing Machine expresses an algorithm.' We will however take a loser approach to the analysis and define informally an **algorithm** to be: A sequence of instructions to do a task (or a problem) with the properties:

a Is of finite length,
b Is executed in steps
c Each step is effective (finite time, can be done by automaton)
d Is runs in finite time (i.e. it terminates)

A **procedure** is similarly defined to be an algorithm that does not necessarily terminates. The language we use to express these instructions is called 'algorithmic language', or sometimes 'Pidgin Pascal'.

The time and space complexities of an algorithm should still have some relation of course to the time and space complexities of a program used to implement that algorithm. The time complexity can be defined as the the total number of times that any (or all) instruction is executed. An instruction here is an algorithmic instruction, i.e. in algorithmic language. Still, this is however an impossible task as most algorithms are a mix of operations. We will count therefore the number of times that a 'key-and-basic'-operation is executed. An operation 'q' is **key-and-basic** in an algorithm A if the total number of operations executed, as well as the total time involved, is roughly proportional to the total number of operations 'q', or, more precisely, if the total number of operations by algorithm A on data of size n is in the same order as the total number of 'q' operations, $\Theta($ total number of 'q'-operations on that input of size n). Again such a key-and-basic operation is not unique, as there are probably control variables, control structures etc. that all could be taken key-and-basic. The metric must however be realistic, so that you take an operation as key-and-basic that reflects the problem at hand: comparison for searching/sorting, scalar multiplications for matrix-multiplications, traversing a link in a graph-traversal, etc. Notice however that using asymtotics to describe the complexities of algorithms, we have indeed found a way to incorporate the uncertainties that are inherent in the theory: The time complexity of any program that is a faithful implementation of the algorithm, in any language, with any compiler, on any machine, and with any load, will have a timecomplexity in the same asymtotic order.

A rather formal way to define the complexity of an algorithm without resorting to a key-and-basic operation is as follows. Let T(A,L,M) be the total run time for algorithm A if it were implemented with language L and if it were run on Turing machine M. Then the complexity class of algorithm A is $O(T(A,L_1,M_1)) \cup O(T(A,L_2,M_2)) \cup O(T(A,L_3,M_3)) \cup \ldots$. Call this complexity class a, then the complexity of A is said to be f, if a=O(f). This definition is not as bad as it appears at first glance, as it can be shown that $O(T(A,L_1,M_1))=O(T(A,L_2,M_2))$ for any two general purpose languages L_i and any two Turing machines M_i, i=1,2.

Definition One can classify algorithms now like this: Algorithm A is said to be **of at most linear/ quadratic/** etc **in best case**, if the function $T_A^{best}(n)$ is such.

The same of course for average case and worst case. Be forewarned however, that it is common to say that a certain algorithm is O(some function), when it really meant that this algorithm is Θ(some function).

How to find performance measures for algorithm A?

1 Find a key-and-basic operation q.
2 Suppose data of size n is to be worked on. Find all possible formations U of this data (relevant to the problem): Let $U=\{I_1, I_2, I_3, \ldots\ldots\}$.
3 Find the probability that this formation actually will occur: $p[I_i]$. Assume all these probabilities are non-zero, remove those from U if needed.
4 Compute $t_A(I_i)$, the number of key-and-basic operations for input I_i by this algorithm A.
5 $T_A^{best}(n)$ = minimum over all i of $\{ t_A(I_i) \}$
$T_A^{average}(n)$ = summation over all i of $p[I_i].\, t_A(I_i)$
$T_A^{worst}(n)$ = maximum over all i of $\{ t_A(I_i) \}$

Note: $T_A^{best}(n) \le T_A^{average}(n) \le T_A^{worst}(n)$, for all n.

1 Searching Through Linear Structures

(This was last updated: Jan 21, 2014, and printed September 8, 2014)

1.1 Linear Searching for an element 'x' in an Unsorted List

Given an unsorted list, what is the complexity for finding an element on this list, given that with probability p the item is on the list on the first place, and given that the probability of finding this element at a particular location is uniformly distributed, that is, Pr[x at location k] $= \frac{1}{n}$, for all $k = 1 \ldots n$.

```
algorithm LinearSearch ($MyA, n$)
(* Assume that the Array is indexed from 0 through $n-1$*)

int k=0   // $k$ is a local integer variable, initialized to the value 0
$MyA[\ n\ ] \leftarrow x$  //Place $x$ as sentinel value at the end of the array
while ($MyA[\ k{+}{+}\ ] \neq x$) do endwhile
if ($k < n$)
    then return ($k-1$)
    else return ($-1$)                // $x$ is not on the List
end LinearSearch
```

This is one of the very few situations where the timing distribution can be given explicitly, so we might as well present it: $\Pr[T(n) = k] = \frac{p}{n}$ for $k = 1, \ldots, n-1$, and $\Pr[T(n) = n] = \frac{p}{n} + (1-p)$. The best, worst, and average time complexities are as follows:

Best Case The best case, $T^B(n) = 1$, occurs obviously when the element is at the first location of the list. The probability of this happening is $\Pr[T(n) = 1] = \Pr[\text{item x is at location } 1] = \frac{p}{n}$.

Worst Case The worst case, $T^W(n) = n$, occurs when either the element is at the last location, or the element is not on the list. The probability of this happening is $\Pr[T(n) = n] = \Pr[\text{item x is at location } n$ *or* item is not on the list$] = \frac{p}{n} + (1-p)$.

Average Case The average case needs perhaps an elaboration:

$$\begin{aligned}
T^A(n) = &\ \Pr[\text{item is on the list}]T^A(n|\text{item is on the list}) \\
&+\Pr[\text{item is } \textit{not} \text{ on the list}]T^A(n|\text{item is } \textit{not} \text{ on the list}) \\
= &\ p\, T^A(n|\text{item is on the list}) + (1-p)\, T^A(n|\text{item is } \textit{not} \text{ on the list}) \\
= &\ p \sum_{k=1}^{n} \Pr[\text{x at location } k] T^A(n|\text{x at location } k) + (1-p)\, n \\
= &\ p \sum_{k=1}^{n} \frac{1}{n}\, k \qquad + (1-p)\, n \qquad = \qquad p\, \frac{1}{n}\, \frac{n\,(n+1)}{2} + (1-p)\, n \\
= &\ (1 - \frac{p}{2})\, n + p = (1 - \frac{p}{2})\, n + o(n)
\end{aligned}$$

Of course, the probability that the timing function is exactly its average value is zero, as the average value is not even an integer (in general). Notice that the asymptotic behavior of the average is very much influenced by the value of p: If $p = 0$, then $T^A(n) = n$, whereas if $p = 1$, then $T^A(n) = \frac{1}{2}\, n + 1$.

$$\boxed{\begin{aligned} T^B(n) &= 1 \\ T^W(n) &= n \\ T^A(n) &= (1 - \tfrac{p}{2})\, n + p = (1 - \tfrac{p}{2})\, n + o(n) \end{aligned}} \qquad (1)$$

Discussion Be sure the answers are correct by checking with your intuition: What if $p = 1$? What if $p = 0$? Does the best case result hold when $p = 0$, why? Can other performance measures, like standard deviation or variance, be computed? What about assumptions we made about the list not being sorted? about the equal probability that each element is searched for? Can we sort the list (by value of the element)? Can we sort by probability? How can we improve performance what chosing a different datastructure, and/or different algorithm?

1.2 Linear Searching for an element 'x' in a Sorted List

Given an sorted list, what is the complexity for finding an element on this list, given that with probability p the item is on the list, and that the conditional probability of finding this element at a particular location is uniformly distributed, that is, Pr[x at location k] $= \frac{1}{n}$, for all $k = 1 \ldots n$, if the element is on the list. Similarly, if the element is not on the list, we assume that the search element is uniformly distributed over the gaps, Pr[x in gap k] $= \frac{1}{n+1}$, for all $k = 1 \ldots n+1$,given it is not on the list.

```
algorithm LinearSearchSortedArray ($MyA, n$)
(* Assume that the Array is indexed from 0 through $n-1$*)
(* Assume also that the Array is sorted in increasing order *)

int $k = -1$                                   // $k$ is a local integer variable, initialized to the value
$-1$
$MyA[\ n\ ] \leftarrow x$ //Place $x$ as sentinel value at the end of the array
while ($MyA[\ {+}{+}k] < x$) do endwhile
if ($k < n$)
    then return ($k$)
    else return ($-1$)                 // $x$ is not on the Array
end LinearSearchSortedArray
```

The explicit probability mass function is now $\Pr[T(n) = k] = \frac{p}{n} + \frac{1-p}{n+1}$ for $k = 1, \ldots, n-1$, and $\Pr[T(n) = n] = \frac{p}{n} + 2\ \frac{1-p}{n+1}$. The usual time complexities are as follows:

Best Case The best case, $T^B(n) = 1$, occurs obviously when the element is at the first location of the list. The probability of this happening is $\Pr[T(n) = 1] =$ Pr[item x is at location 1] $= \frac{p}{n}$.

Worst Case The worst case, $T^W(n) = n$, occurs when either the element is at the last location, or the element is not on the list, and it is larger than the element at location 'n-1'. The probability of this happening is $\Pr[T(n) = n] =$ Pr[item x is at location n *or* item is not on the list and larger than element n-1] $= p\ \frac{1}{n} + (1-p)\ \frac{2}{n+1}$.

Average Case The average case needs to be elaborated on:

$$\begin{aligned}
T^A(n) = \quad & \Pr[\text{item is on the list}]T^A(n|\text{item is on the list}) \\
& + \Pr[\text{item is } \textit{not} \text{ on the list}]T^A(n|\text{item is } \textit{not} \text{ on the list}) \\
= \quad & p\ T^A(n|\text{item is on the list}) + (1-p)\ T^A(n|\text{item is } \textit{not} \text{ on the list}) \\
= \quad & p \sum_{k=1}^{n} \Pr[\text{x at location } k] T^A(n|\text{x at location } k) \\
& + (1-p) \sum_{k=1}^{n+1} \Pr[\text{x at gap } k] T^A(n|\text{x at gap } k) \\
= \quad & p \sum_{k=1}^{n} \frac{1}{n}\ k + (1-p)\{\sum_{k=1}^{n} \frac{1}{n+1} T^A(n|\text{x at gap } k) + \frac{1}{n+1} T^A(n|\text{x at gap } n+1)\} \\
= \quad & p\ \frac{1}{n}\ \frac{n\ (n+1)}{2} + (1-p)\ \frac{1}{n+1}\{\sum_{k=1}^{n} k + n\}
\end{aligned}$$

$$= p\frac{1}{n}\frac{n\,(n+1)}{2} + (1-p)\,\frac{1}{n+1}\{\frac{n\,(n+1)}{2} + n\}$$
$$= \frac{1}{2}\,(n+p) + (1-p)\,\frac{n}{n+1} = \frac{1}{2}\,n + o(n)$$

Notice that the asymptotic behavior is no longer influenced by the value of p: $\frac{1}{2}\,n + \frac{1}{2} \leq T^A(n) \leq \frac{1}{2}\,n+1$ for alll values of p.

Discussion Be sure the answers are correct by checking with your intuition: What if $p = 1$? What if $p = 0$? Does the best case result hold when $p = 0$, why? Can other performance measures, like standard deviation or variance, be computed? What about the equal probability that each element is searched for? Can we sort by probability? How can we improve performance what chosing a different datastructure, and/or different algorithm? In particular, can the worst case behavior be improved upon?

$$\boxed{\begin{aligned} T^B(n) &= 1 \\ T^W(n) &= n \\ T^A(n) &= \tfrac{1}{2}\,(n+p) + (1-p)\,\tfrac{n}{n+1} = \tfrac{1}{2}\,n + o(n) \end{aligned}} \qquad (2)$$

1.3 Binary Searching for an element 'x' in a Sorted Array

Now that we have direct access to all the elements in an array, we will now try to eliminate the largest sub-array from consideration. Since there are a high number of slightly different algorithms for this case, we will present one for discussion:

```
algorithm BinarySearch (Array, m, n)
(* Precondition: $m \leq n$ and $A[i] < A[i+1]$, $i = m, \ldots, n-1$ *)

while ($m \leq n$) do
    $middle := \lfloor \frac{m+n}{2} \rfloor$
    case ($x < A[middle]$) then  $n := middle - 1$
    case ($x = A[middle]$) then  RETURN ($middle$)
    case ($x > A[middle]$) then  $m := middle + 1$
    end-while
RETURN ( 0 )
end BinarySearch
```

The 'switch' statement could wreak havock on our well-intentioned algorithm, that would say: all three 'cases' are obtained at the same cost. However, every compiler takes the switch statement, and replaces it with a sequence of `if elif elif end-if` and the analyze for the best- and worst case complexities may be different. Please do so for the following two versions:

```
algorithm BinarySearch3a (Array, m, n)
(* Precondition: $m \leq n$ and $A[i] < A[i+1]$, $i = m, \ldots, n-1$ *)

while ($m \leq n$) do
    $middle := \lfloor \frac{m+n}{2} \rfloor$
    if ($x < A[middle]$) then  $n := middle - 1$
    elif ($x = A[middle]$) then  RETURN ($middle$)
    elif ($x > A[middle]$) then  $m := middle + 1$
    end-while
RETURN ( 0 )
```

```
end BinarySearch

algorithm BinarySearch3b (Array, m, n)
(* Precondition: $m \leq n$ and $A[i] < A[i+1]$, $i = m, \ldots, n-1$ *)

while ($m \leq n$) do
        $middle := \lfloor \frac{m+n}{2} \rfloor$
        if ($x > A[middle]$) then  $n := middle + 1$
        elif ($x = A[middle]$) then  RETURN ($middle$)
        elif ($x < A[middle]$) then  $m := middle - 1$
        end-while
RETURN ( 0 )
end BinarySearch
```

Finally, compare this with the complexities for the 'easiest' case, where we just split the array in two:

```
algorithm BinarySearch2 (Array, m, n)
(* Precondition: $m \leq n$ and $A[i] < A[i+1]$, $i = m, \ldots, n-1$ *)

while ($m < n$) do
        $middle := \lfloor \frac{m+n}{2} \rfloor$
        if ($x > A[middle]$) then  $m := middle + 1$
        else $n := middle$
        end-while
if $x = A[m]$ then RETURN (m) else RETURN (0)
end BinarySearch2
```

First convince yourself that this algorithm is correct (i.e. correct the mistake that may be there for special values of m and n), and analyze again the algorithm for best - and worst case complexities.

THE LANGUAGE OF ASYMPTOTICS

The big-*O* symbol was introduced by Bachmann-Landau in 1927, and now widely used in the studies on asymptotic behavior of functions. It was introduced as a means to suppress secondary information in an asymptotic region and needed only as a binary operator to indicate the relative growth of two functions. The same symbol is now also used for classification of functions and classification of families of functions. All this has the big advantage that, once you understand its function and meaning, you can get a little careless in using it, as there will always be a definition to 'cover' it. The disadvantage is that if you do not know its meaning, you tend to be even more confused because you see it being used in a different meaning, quite often on the same page, without the author(s) warning you of it's extended definition. Add to this the fact that different books may have slightly different and conflicting definitions and the chaos is almost complete.

In the studies of algorithm-complexity, both for obtaining upper-bounds as well as lower-bound theory, only functions from the non-negative integers to the positive reals need to be considered, so unless otherwise stated, all functions to be considered in this paper belong to the universe **U** of functions from N to R^+. Also, the asymptotic region for our purposes is the neighborhood of infinity, i.e. the interval (N_0,∞) for some number N_0 conveniently big enough. The commonly used definitions in Computer Science stem from [KNUT 76], but in particular the big omega, Ω, is under heavy discussion, see e.g. [BRAS 85]. In the following definitions we will therefore not make much use of this omega. It should be mentioned that the book by Wilf gives an excellent introduction to the concepts, [WILF86]. Also note that the definitions and theorems given here is only for the asymptotic behavior in the neighborhood of $+\infty$, and that the domain from which the functions are drawn includes *only nonnegative functions*.

ORDERS OF MAGNITUDE AS BINARY OPERATORS: BIG O.

In comparing two functions, we might learn from the comparison of two numbers and the rules governing these comparisons. De Bruijn uses this technique eloquently in his book: Suppose you are typing a little overview of the properties of ≤ on say the positive real numbers, but your typewriter is rather old and does not have the symbol '≤' on it. You may therefore decide to write 'a=L(b)' if you realy mean 'a≤b'. Throughout your essay, you will have statements involving this '=L( )' combination where the = symbol is not meant 'equality', but has only meaning in combination with the 'L': the ≤-remains a binary operator. For example, you can expect to say things like

- 3=L(5)
- If 3=L(5) and 11=L(13) then 3+11=L(5+13)
- If a=L(b) and b=L(c) then a=L(c)

But you would not see things that 'do not make sense', like

- L(5)=3
- If 3=L(5) and 11=L(13) then 3+11=L(5)+L(13)
- If a=L(b) and b=L(c) then a=L(L(c)),

In the design and analysis of algorithms, one often needs to compare two functions, each representing the complexity of an algorithm, and the question remains: Which function is better? So instead of comparing two numbers, one now needs to compare two sets of numbers and, to make matters even worse, the actual function values are often not important. (Remember that the functions are language and machine independent?). In this case, one wants to look therefore at the growth of the function as an indicator of how big the problem size a particular algorithm can be. So one kind of naturally looks at the limit of these functions at infinity. But, by the nature of these functions, this limit does not exist: these functions themselves diverge to infinity. It is for this reason that one considers the relative growth of these functions. But what notation should be used for the comparison of the behavior of these two function in the neighborhood of their asymptote? The same situation can be envisioned as in the introduction where =L(.) was introduced since the typewriter didn't have '≤'. In this case, the typewriter has ≤, but it is inappropriate. Rather than introducing a new symbol, say $\propto$, it is common to use '=O(.)' to stand for a binary operator, and it should not be interpreted as 'left of the = symbol' is equal to 'the right is the = symbol'.

BIG OH.

Definition: Let T and f be two functions in **U**, then *T= O(f)* if there is a natural number N, so that there is a constant *A* in the positive reals (which may depend on the chosen N) so that for all natural numbers n larger than N is true that T(n) ≤ *A* f(n).

Notice, that this definition is very much like the definitions as usually given in calculus (continuïty, differentiability, etc.). A more workable equivalents for this big *O* is in theorem:

Theorem: If f(n)>0 for all n and the limit $\lim\limits_{n\to\infty} \frac{T(n)}{f(n)}$ exists and is finite, then T=O(f).

Proof: Left as an Exercise.

Remark: The reverse is not true: If $f(n)>0$ for all n and if $T=O(f)$, the limit of T/f may not exist. Take e.g. $T(n)=1$ for n even and $T(n)=2$ for n odd, while $f(n)=3$ for all n, then the limit does not exist, while $T=O(f)$. One could make above theorem an if-and-only-if relation by introducing the 'limsup', which is beyond the scope of this course. Most of the cases that will be investigated in the analysis of algorithms deal with functions where the above theorem can be used, even though sometimes L'Hôpital's rule need to be applied.

Theorem: The following statements a-d all hold and part e is a counterexample showing that part d can not be reversed, the proofs are again left as an exercise.

a $c.f = O(f)$ and $f=O(c.f)$ for every positive constant c.
b $n^k = O(n^{k+x})$ for all positive k and x.
c $p_q(n)= O(n^q)$, for any polynomial of degree q.
d $n^k = O(c^n)$ for $c>1$ and $k\geq 0$.
e $c^n \neq O(n^k)$ for $c>1$ and $k\geq 0$.

The =O notation indicates indeed 'bounded above by a constant multiple of'. If you do not feel comfortable with the O-notation, you may want to write the equivalent statements for numbers using '=L'. You will find that this can not be done for part a).

Definition Functions can be classified according to their behavior when n grows large:
A function T(n) is said to be **of at most logarithmic growth** if $T(n)= O(\log n)$,
A function T(n) is said to be **of at most linear growth** if $T(n)= O(n)$,
A function T(n) is said to be **of at most quadratic growth** if $T(n)= O(n^2)$,
A function T(n) is said to be **of at most polynomial growth** if $T(n)= O(n^k)$, for some natural number $k \geq 1$.
A function T(n) is said to be **of at most exponential growth** if there is a constant c, such that $T(n)= O(c^n)$, and $c>1$.
A function T(n) is said to be **of at most factorial growth** if $T(n)= O(n!)$.

The relation $\leq$ between numbers is thus extended in a rather natural way to a relation =O( ) between growth of functions.

BIG THETA.

As these functions indicate, there are an enormous amount of different functions f, such that $T=O(f)$ for a given function T. For instance, a function T that is of at most linear growth, is also of at most quadratic growth and also of at most cubic growth, etcetera. This is just like there are an infinite number of positive real numbers p, such that $5=L(p)$. For numbers you would therefore want to find the smallest such p for which the above holds, and define 'equality': $a=E(b)$ if both $a=L(b)$ and $b=L(a)$. For our functions, we will do something similar:

Definition Two functions T and f are said to be of equal growth, $T=\Theta(f)$ if-and-only-if both $T=O(f)$ and $f=O(T)$.

Even then, the number of functions f such that $T= \Theta(f)$ for a given T, is still large. For instance, $T(n)=n+\sqrt{n}$ is in the same class of functions as $2n+5\sqrt{n}$, $n\sqrt{2}+\log n$, $n+\lg n + \lg \lg n$, etc. It is customary to take the simplest such f, like $f(n)=1$, $\log n$, $\log \log n$, n, $n \log n$, $n.\sqrt{n}$, n^2, c^n, 2^n, $n!$, etcetera. It takes only a minute of reflection that this is also the situation with numbers: 5, 10/2, 25/5, 4.999..., 5.0000..., ... are all equal and usually we write just 5 as the simplest representation, and can be regarded as a representation out of an equivalence class. Simplest in this respect is intended to mean: conceptually simplest, rather than computationally simplest.

Functions can now also be classified according to their class:
A function T(n) is said to be **of logarithmic growth** if $T(n)= \Theta(\log n)$,
A function T(n) is said to be **of linear growth** if $T(n)= \Theta(n)$,
A function T(n) is said to be **of quadratic growth** if $T(n)= \Theta(n^2)$,
A function T(n) is said to be **of polynomial growth** if $T(n)= \Theta(n^k)$, for some $k\geq 1$.
A function T(n) is said to be **of exponential growth** if there is a $c>1$, such that $T(n)= \Theta(c^n)$.
A function T(n) is said to be **of factorial growth** if $T(n)= \Theta(n!)$.

OTHER ASYMPTOTIC LANGUAGE.

So far, we have introduced the equivalent of the $\leq$ symbol for asymptotic behavior. Other comparisons between numbers can be made, $<, >, \geq, =$, etc. and similarly, other symbols for asymptotic studies can be introduced. Rather than doing this 'long' way, let us use the limit of the ratio's as defining these asymptotics, keeping in mind that this defenition is weaker than the normal one, but acceptable for most computer science applications. (Our definition applies only to pairs of functions for which the limit of the ratio exists. If this is not the case, then the 'lim' must be replaced by either 'limsup' or 'liminf', depending on the actual definition. These always exist if infinity is allowed as a limit point.)

Definition Let T and f be functions in **U**, such that the limit (limsup) of the ratio T(n)/f(n) exists (but possibly infinity) as n goes to infinity. Call this limit a. Note, that $a \geq 0$, since all functions involved are non-negative. Note also, that the limit sometimes diverges, we will say then that the limit 'exists, but is ∞'. Then

T=o(f)	if $a = 0$	"<"
T~f	if $a = 1$	"~"
T=O(f)	if a is finite, i.e. $a < \infty$	"≤"
T=Ω(f)	if a is strictly positive, possibly infinite, i.e. $0 < a \leq \infty$	"≥"
T=Θ(f)	if a is strictly positive and finite, i.e. $0 < a < \infty$	"≈"
T=ω(f)	if a is infinite, $a = \infty$	">"

In all these cases, the function T is more or less approximated by a function f, where 'more or less' reflects the behavior in the asymptotic region. As such, the asymptotically equal, ~, retains the most information and is the most precise approximation. The o is more precise than O since we know the exact limit, rather than only being finite, etc.
The big omega Ω plays now the role for growth behavior that ≥ plays for numbers: a≥b means: a is at least as big as b. The classification of functions and algorithms can also be extended by saying something like if T=Ω(f), then T is at least as big as f, or, f is a lowerbound for T:

Definition A function T(n) is said to be **at least linear** (etc) if T(n)= Ω(n),
Moreover, a function T(n) with T=Θ(f) is indeed both at least and at most of linear growth.

Notice, that a function T(n) does <u>not</u> have to be linear itself in order to be of linear growth, it just has to be between two linear functions, e.g. T(n)=(2-sin n)n is between $f_1(n)=n$ and $f_2(n)=3n$. The same holds true of course for polynomial, exponential etc.

Notice also, that these classifications does not 'cover' all functions, just the more common ones. E.g. there are functions that grow slower than logarithmic (log log n). A more important example are the functions whose growth-rate is between polynomial and exponential, they are said to be **of subexponential growth**: they grow faster than n^k for any finite k, yet grow slower than c^n for every c>1. Examples: 2 to the power of log n, 2 to the power of √n, and functions like this. Also, there are functions that grow faster yet, the super exponential functions: 2 to the power of n-squared, 2 to the power of 2 to the power of 2, etc.

ASYMPTOTIC NOTATION AS UNARY OPERATOR

Let us go back to our =L situation. After a while writing/reading an essay with this kind of notation instead of ≤, you feel comfortable in using it and all of a sudden you find yourself writing statements like:

- L(3)=L(5) to mean 'something that is less or equal to' 3 is less than or equal to 5
- 7=3+L(5) to mean 7 is less or equal to 3 plus something less or equal to 5.
- L(3)+L(5)=L(8) to mean 'something that is less or equal to' 3 added to 'something that is less or equal to' 5 is itself 'something less or equal to' 8.
- 3=L(5)+L(8) to mean 3 is less or equal to ...?

All of a sudden, the =L convention is no longer valid, but yet you feel comfortable in using it. The use of L(.) is thus extended to the following:

- An expression that involves L as a unary symbol is to be considered as a class of numbers (a set). For example L(3)+L(5) stands for the set of numbers a+b, so that a=L(3) and b=L(5).
- The =-symbol becomes now either the or the symbol, depending on the left-hand side: If the left hand side is still a number, it becomes the symbol -, whereas if the left hand side becomes a set itself, it should be read as ⊆.
- If the 'argument' of L is a set itself, like L(3+L(5)), then the interpretation is the set of numbers less than or equal to '3 plus something less or equal to 5'.

Note however, that in both cases the = is not symmetric: L(3)+L(5)=L(2)+L(7), but it is <u>not</u> true that L(2)+L(7)=L(3)+L(5).

The use of O is extended in the same manner. O(f) can thus also mean: The set of functions T, such that there is N_0 and there is a constant A such that .... or alternately (if these limits exist):

$O(f)$ is the set of functions T, such that $\lim T(n)/f(n)$ is finite.
$\Theta(f)$ is the set of functions T, such that $\lim T(n)/f(n)$ is finite and strictly positive.
$o(f)$ is the set of functions T, such that $\lim T(n)/f(n)$ is zero.
$\omega(f)$ is the set of functions T, such that $\lim T(n)/f(n)$ is infinity
$\Omega(f)$ is the set of functions T, such that $\lim T(n)/f(n)$ is strictly greater than zero, possibly infinity.

There are a number of complimentary relations amongst these sets:
Theorem For every T in $\mathbf{U}$, $O(T) \cap \Omega(T) = \Theta(T)$.
Theorem For every T in $\mathbf{U}$, $O(T) - \Theta(T) = o(T)$.
Theorem For every T in $\mathbf{U}$, the union of $O(T)$ and $\omega(T)$ is $\mathbf{U}$, while their intersection is empty. The same relationship exists between $o(T)$ and $\Omega(T)$.

The rules for expressions involving O's thus needs to be extended to cover both unary as well as binary operator meanings:

- An expression that involves O as a unary symbol is to be considered as a class of functions (a set). For example $O(f)+O(g)$ stands for the set of functions $(T_1+T_2)(n)$, so that $T_1=O(f)$ and $T_2=O(g)$.
- The =-symbol becomes now either the - tor the $\subseteq$ symbol, depending on the left-hand side: If the left hand side is still a number, it becomes the symbol -, whereas if the left hand side becomes a set itself, it should be read as $\subseteq$.
- If the 'argument' of O is a set itself, like $O(f+O(g))$, then the interpretation is the set of functions bounded above by f+a function which is bounded above by g.

It is therefore quite common to see expressions like $O(n)+O(\log n)=O(n)$ and such. With the big-O as sets, note that both

$$O(1) = O(n) = O(n^2) = O(n^3) = O(n^4) = O(n^5) = O(n^9)\ldots \text{ and}$$
$$O(1) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(n^4) \subseteq O(n^5) \subseteq O(n^9)\ldots$$

become acceptable notation. A theorem with some of the most common combinations is:

Theorem If $T_1=O(f_1)$, $T_2=O(f_2)$ and $f_1=O(g)$, where all functions are from N to the positive reals, then

a	f	$= O(f)$
b	$c\, O(f)$	$= O(f)$
c	$O(f)+O(f)$	$= O(f)$
d	$O(O(f))$	$= O(f)$
e	$O(f)+O(g)$	$= O(\max(f,g))$
f	$O(f).O(g)$	$= O(f\, g)$
g	$O(f.g)$	$= f\, O(g)$
h	$O(f+g)= O(\max(f,g))$	

Similarly, Θ is abused in exactly the same way, except that there is the added convenience that a certain symmetry is preserved.

Theorem If f and g are functions from N to the positive reals, then

a	f	$= \Theta(f)$
b	$c\, \Theta(f)$	$= \Theta(f)$
c	$\Theta(f)+ \Theta(f)$	$= \Theta(f)$
d	$\Theta(\Theta(f))$	$= \Theta(f)$
e	$\Theta(f)+ \Theta(g)$	$= \Theta(\max(f,g))$
f	$\Theta(f)\,\Theta(g)$	$= \Theta(f\, g)$
g	$\Theta(f\, g)$	$= f\, \Theta(g)$

Some examples:

Example 1: $6(1^2 + 2^2 + 3^2 + \ldots\ldots + n^2) = 2n^3 + 3n^2 + n = 2n^3 + 3n^2 + O(n)$
$= 2n^3 + O(n^2 + n) = 2.n^3 + O(n^2) = O(n^3) = O(n^4)$.

Example 2: $6(1^2 + 2^2 + 3^2 + \ldots\ldots + n^2) = 2n^3 + 3n^2 + n = 2n^3 + 3n^2 + \Theta(n)$
$= 2n^3 + \Theta(n^2 + n) = 2n^3 + \Theta(n^2) = \Theta(n^3)$

Example 3: $\sqrt[n]{n} = 1 + (\ln n)/n + O((\ln n)^2/n^2)$
$n \times (\sqrt[n]{n} - 1) = \ln n + O((\ln n)^2/n)$.

Example 4: Harmonic numbers:

Alternate definition

Definitions:

A function $f: \mathcal{R} \to \mathcal{R}$ or $f: \mathcal{N} \to \mathcal{R}$ is ***bounded*** if there is a constant k such that $|f(x)| \le k$ for all x in the domain of f.

For functions $f, g: \mathcal{R} \to \mathcal{R}$ or $f, g: \mathcal{N} \to \mathcal{R}$ (sequences of real numbers) the following are used to compare their growth rates:

- f is ***big-oh*** of g (g ***dominates*** f) if there exist constants C and k such that $|f(x)| \le C|g(x)|$ for all $x > k$.

  Notation: f is $O(g)$, $f(x) \in O(g(x))$, $f \in O(g)$, $f = O(g)$.
- f is ***little-oh*** of g if $\lim_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| = 0$; i.e., for every $C > 0$ there is a constant k such that $|f(x)| \le C|g(x)|$ for all $x > k$.

  Notation: f is $o(g)$, $f(x) \in o(g(x))$, $f \in o(g)$, $f = o(g)$.
- f is ***big omega of*** g if there are constants C and k such that $|g(x)| \le C|f(x)|$ for all $x > k$.

  Notation: f is $\Omega(g)$, $f(x) \in \Omega(g(x))$, $f \in \Omega(g)$, $f = \Omega(g)$.
- f is ***little omega of*** g if $\lim_{x\to\infty} \left|\frac{g(x)}{f(x)}\right| = 0$.

  Notation: f is $\omega(g)$, $f(x) \in \omega(g(x))$, $f \in \omega(g)$, $f = \omega(g)$.
- f is ***theta of*** g if there are positive constants C_1, C_2, and k such that $C_1|g(x)| \le |f(x)| \le C_2|g(x)|$ for all $x > k$.

  Notation: f is $\Theta(g)$, $f(x) \in \Theta(g(x))$, $f \in \Theta(g)$, $f = \Theta(g)$, $f \approx g$.
- f is ***asymptotic*** to g if $\lim_{x\to\infty} \frac{g(x)}{f(x)} = 1$. This relation is sometimes called ***asymptotic equality***.

  Notation: $f \sim g$, $f(x) \sim g(x)$.

Facts:

1. The notations $O(\ )$, $o(\ )$, $\Omega(\ )$, $\omega(\ )$, and $\Theta(\ )$ all stand for *collections* of functions. Hence the equality sign, as in $f = O(g)$, does not mean equality of functions.

2. The symbols $O(g)$, $o(g)$, $\Omega(g)$, $\omega(g)$, and $\Theta(g)$ are frequently used to represent a typical element of the class of functions it represents, as in an expression such as $f(n) = n \log n + o(n)$.

3. *Growth rates*:
- $O(g)$: the set of functions that grow no more rapidly than a positive multiple of g;
- $o(g)$: the set of functions that grow less rapidly than a positive multiple of g;
- $\Omega(g)$: the set of functions that grow at least as rapidly as a positive multiple of g;
- $\omega(g)$: the set of functions that grow more rapidly than a positive multiple of g;
- $\Theta(g)$: the set of functions that grow at the same rate as a positive multiple of g.

4. Asymptotic notation can be used to describe the growth of infinite sequences, since infinite sequences are functions from $\{0, 1, 2, \ldots\}$ or $\{1, 2, 3, \ldots\}$ to $\mathcal{R}$ (by considering the term a_n as $a(n)$, the value of the function $a(n)$ at the integer n).

5. The big-oh notation was introduced in 1892 by Paul Bachmann (1837–1920) in the study of the rates of growth of various functions in number theory.

6. The big-oh symbol is often called a *Landau symbol*, after Edmund Landau (1877–1938), who popularized this notation.