PROJECT TITLE: **Mathematics of the Future: Understanding and Application of Mathematics with the help of Technology**

Intellectual Output 3: FutureMATH video collection

Programme: Erasmus+

Key Action: Cooperation for innovation and the exchange of good practices

Action Type: Strategic Partnerships for higher education

Ref. No.: 2020-1-RS01-KA203-065388

Author: Rale Nikolić

Analysis of Complexity of algorithms

Motivation.

- Since there are often many possible algorithms or programs that compute the same results, we would like to use the one that is fastest.
- How do we decide how fast an algorithm is? Since knowing how fast an algorithm runs for a certain input does not reveal anything about how fast it runs on other inputs, we need another measure that tells us how fast it is for any input. A formula that relates input size to the running time of the algorithm satisfies this requirement.
- We also want to ignore machine dependent factors. If an algorithm takes two seconds on one machine for a given input, a trivial way to get it to run in one second is to use a machine that is twice as fast. There is a constant multiplicative factor relating the speed of an algorithm on one machine and its speed on another, which we will ignore.
- We are only interested in how fast an algorithm runs on large inputs, since even slow algorithms finish quickly on small inputs.

Initial problem

When studying the complexity of an algorithm, we are concerned with the growth in the number of operations required by the algorithm as the size of the problem increases. In order to get a handle on its complexity, we first look for a function that gives the number of operations in terms of the size of the problem, usually measured by a positive integer n, to which the algorithm is applied. We then try to compare values of this function, for large n, to the values of some known function, such as a power function, exponential function, or logarithm function. Thus, the growth of functions refers to the relative size of the values of two functions for large values of the independent variable.

How does one go about analyzing programs to compare how the program behaves as it scales? For example, let's look at a **vectorMax()** function coded in C++:

```
int vectorMax(Vector<int> &v){
        int currentMax = v[0];
        int n = v.size();
        for (int i=1; i < n; i++){
                if (currentMax < v[i]) {
                   currentMax = v[i];
                }
        }
        return currentMax;
}
```

If we want to see how this algorithm behaves as *n* changes, we could do the following:

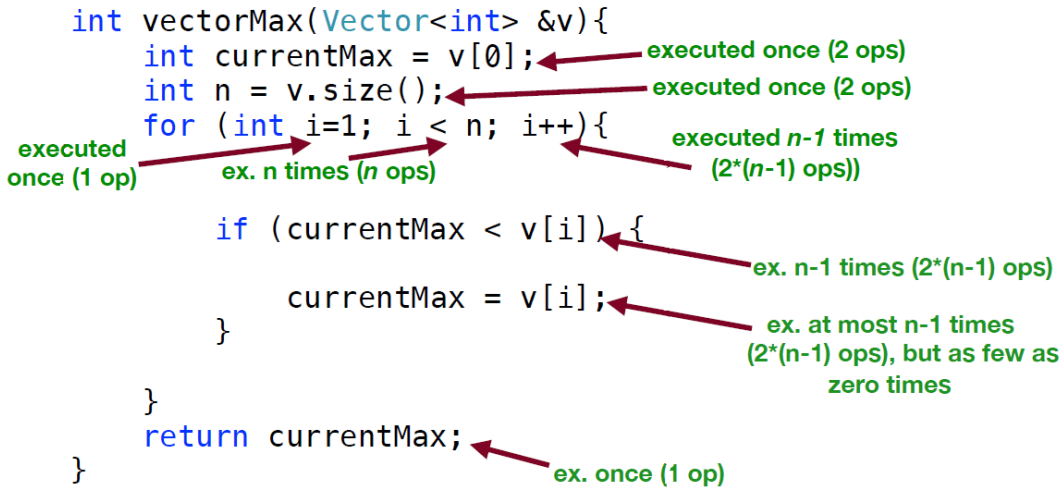
1) Code the algorithm in some program language
2) Determine, for each instruction of the compiled program the time needed to execute that instruction (need assembly language)
3) Determine the number of times each instruction is executed when the program is run.
4) Sum up all the times we calculated to get a running time.

Steps 1) - 4) might work, but it is complicated, especially for today's machines that optimize everything "under the hood." (and reading assembly code takes a certain patience).

Instead of those complex steps, we can define *primitive operations* for our code.

- Assigning a value to a variable
- Calling a function
- Arithmetic (e.g., adding two numbers)
- Comparing two numbers
- Indexing into a Vector
- Returning from a function

We assign "1 operation" to each step. We are trying to gather data so we can compare this to other algorithms.



Primitive operations for **vectorMax()**:

$$\text{at least: } 2 + 2 + 1 + n + 4 \cdot (n-1) + 1 = 5n + 2$$

$$\text{at most: } 2 + 2 + 1 + n + 6 \cdot (n-1) + 1 = 7n$$

i.e., if there are n items in the Vector, there are between $5n + 2$ operations and $7n$ operations completed in the function. Hence, the best case is $5n + 2$ operations, and the worst case is $7n$. Do we *really* need this much detail? No. Let's simplify: we want a "big picture" approach. It is enough to know that **vectorMax()** grows *linearly proportionally to n.* In other words, as the

number of elements increases, the algorithm has to do proportionally more work, and that relationship is linear.

Simplification of the previous procedure is done by using Big-*O* notation. Actually, we use big-*O* notation as a way of simplifying the running time of an algorithm based on the size of its input. The notation allows us to talk about algorithms at a higher level and estimate how implementations of algorithms will behave.

In order to introduce the notion of Big-*O* notation, it is necessary to first recall the notion of the limit of a function.

Functional limits

Definition 1. The limit of function $f: [b, c] \to \mathbb{R}$ as x goes to $a \in [b, c]$ equals L if and only if for every $\varepsilon > 0$ there exists some $\delta > 0$ such that whenever $x(\neq a)$ is within δ of a, then $f(x)$ is within ε of L.

If a is a limit point of the domain of f, then, intuitively, the statement

$$\lim_{x \to a} f(x) = L$$

is intended to convey that values of $f(x)$ get arbitrarily close to L as x is chosen closer and closer to a. The issue of what happens when $x = a$ is irrelevant from the point of view of functional limits. In fact, a need not even be in the domain of f.
The above definition is so called $\delta - \varepsilon$ definition. Actually, if we choose output tolerance $\varepsilon > 0$, than must be some input tolerance $\delta > 0$ so that any input within δ of a has an output within ε of L (see Figure 1).

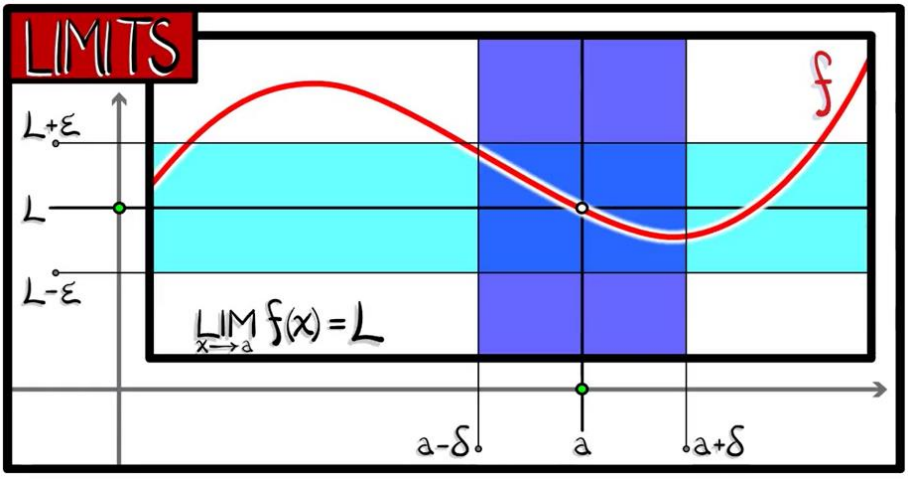


Figure 1.

The "critical" part of above definition is that as you change ε you need to be update δ (see Figure 2). If you make ε smaller still and decrease your level of acceptable error of the output, you need to find some amount of acceptable error on the input. And this has to continue for every possible non-zero value of ε.

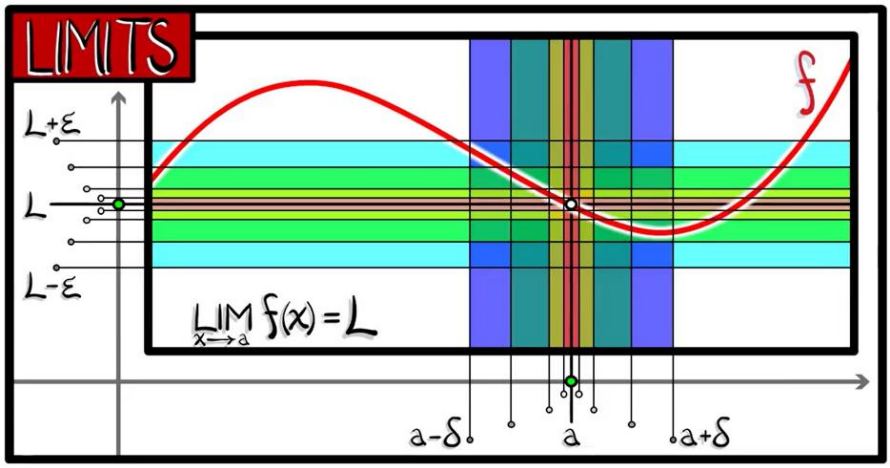


Figure 2.

This view of the definition is extendable in other context. Consider the limit as x goes to infinity of $f(x)$. What does it mean if that limit is equal to L? For given output tolerance ε, there must be some tolerance on the input that guarantees striking within ε of L. In this case there is some bound $M \in \mathbb{R}$, so that whenever your input is greater than M, then your output is within ε of L. As before, this must be true no meter what ε you choose. Furthermore, If you ε smaller and smaller, then M will be larger and larger.

If the limits as x goes to infinity of $f(x)$ exists (see Figure 3), we use next denotation

$$\lim_{x\to\infty} f(x) = L.$$

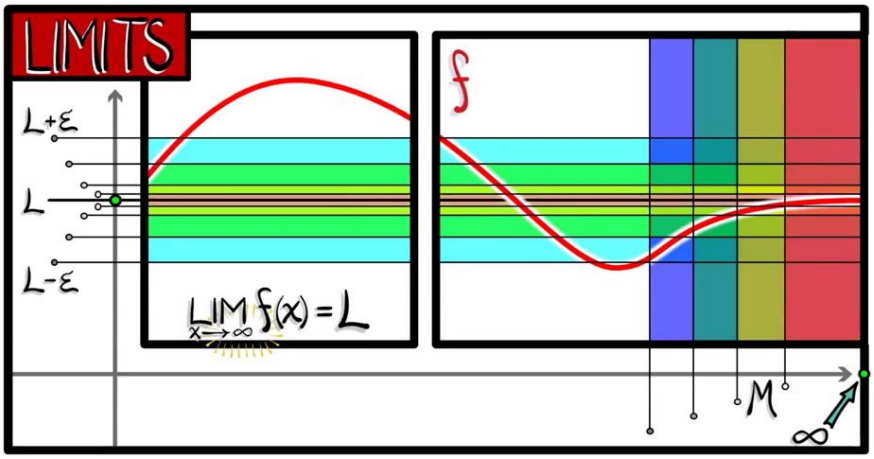


Figure 3.

Analogous holds for limit as x goes to $-\infty$ of $f(x)$.

Big O notation

Now we will talk about orders of growth. Actually, we will talk about what happens when function gets very, very small, or very, very large, i.e. how fast function goes to zero, or how quickly goes to infinity. Quantifying this rates of changes, will give us a new language - Big-*O* notation (or asymptotic notation).

Definition 4. Let functions f and g defined in the neighborhood of the point a, satisfy that

$$\lim_{x\to a} f(x) = \lim_{x\to a} g(x) = \infty.$$

1) Functions f and g are of the same order, when x goes to a, if and only if holds

$$\lim_{x\to a} \frac{|f(x)|}{|g(x)|} = k, (k \neq \infty, k \neq 0).$$

2) Function f has a higher order than function g, when x goes to a, if and only if holds

$$\lim_{x\to a} \frac{|f(x)|}{|g(x)|} = \infty.$$

3) Function f has a lower order than function g, when x goes to a, if and only if holds

$$\lim_{x\to a} \frac{|f(x)|}{|g(x)|} = 0.$$

Remark 1. In terms of introducing a definition for a Big-*O* notation for using it for analysis of complexity of algorithms we will be particularly interested in the case when $a = \infty$ in Definition 4.

In particular, if $k = 1$ in first part of Definition 4, then we say that function f has the same asymptotic behavior as function g, when x goes to a, or that functions f and g are *asymptotically equivalent*, when x goes to a. This will be denoted with

$$f(x) \sim g(x), \text{ when } x \to a.$$

Example 1. It is well-known that

$$\lim_{x\to\infty} \frac{x^3 + 3x^2 + 2x - 5}{x^3} = 1,$$

and

$$\lim_{x\to\infty}\frac{\sqrt{x^2+2x-5}}{x}=1,$$

hold. Then we can write

$$x^3+3x^2+2x-5\sim x^3, \text{ when } x\to\infty,$$

and

$$\sqrt{x^2+2x-5}\sim x, \text{ when } x\to\infty.$$

Now we will apply second (or third) part od Definition 4, to talk about what happens when some classes of functions gets very, very large, when x goes to ∞, i.e. how quickly its go to ∞, when x goes to ∞. Actually, we will compare the growth of three different kinds of functions of x, when x goes ∞:

- power functions $y=x^r$, for $r>0$,
- exponential functions $y=a^x$ for $a>1$,
- logarithmic functions $y=\log_b x$, for $b>1$.

Some examples are plotted in Figure 4.

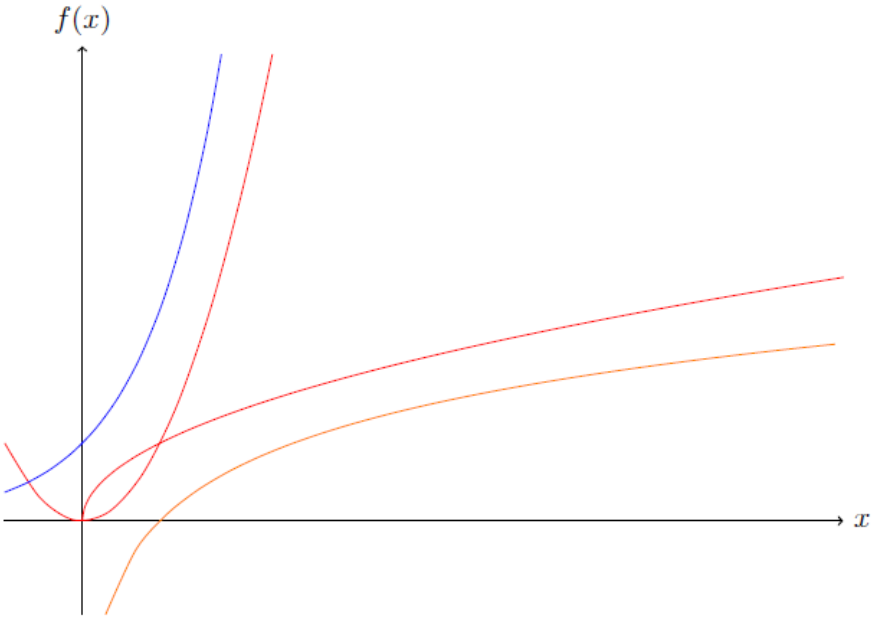


Figure 4. Graph of functions $=\ln x\,, y=\sqrt{x}, y=x^2, y=e^x$.

All power functions, exponential functions, and logarithmic functions (as defined above) go to ∞, when x goes to ∞. But these three classes of functions goes to ∞ at different rates. The main result we want to focus on is the following one. It says e^x grows faster than any power function

while $\log x$ grows slower than any power function. For instance, for functions $y = x$ and $y = \ln x$ holds that

$$\lim_{x\to\infty} \frac{x}{\ln x} = \infty,$$

or equivalent

$$\lim_{x\to\infty} \frac{\ln x}{x} = 0.$$

Accordingly to the Definition 4, then we can say that f has a higher order than g (or equivalent g has a lower order than f), and we denote that $f \prec g$. From previous limits it follows that $\ln x \prec x$, when x goes to ∞. Similarly, from

$$\lim_{x\to\infty} \frac{x^2}{x} = \infty,$$

It follows that $x \prec x^2$, when x goes to ∞, also. Furthermore, the relation $\prec$ is transitive and we have that

$$\ln x \prec x \prec x^2,$$

when x goes to ∞. In this way, we can obtain the next scale of growth

$$\ldots \prec \ln x \prec \cdots \prec \sqrt[3]{x} \prec \sqrt{x} \prec x \prec x^2 \prec x^3 \prec \cdots \prec 2^x \prec e^x \prec 3^x \prec 4^x \prec \cdots$$

when x goes to ∞. This scale are infinite and dense. This means that between any two functions in this scale it is possible inserts infinity many functions. For example, between functions $f(x) = x$ and $g(x) = x^2$, we can insert function $g(x) = x^\alpha$, for every $\alpha \in (1,2)$. In this way, we establish the order of growth for some classes of functions, when x goes to ∞.

Now we will give definition for big-*O.*

Definition 5. Given a function $g: \mathbb{R} \to \mathbb{R}$

$$O(g) = \left\{ f(x) : 0 \lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} < \infty \right\}.$$

$f \in O(g)$ means that f is equal or less order than g.

According to previous definition, if $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} = \infty$ is satisfied, then $f \notin O(g)$.

Remark 2. As we mentioned before, for our needs (to calculate the complexity of the algorithm) we assumed in previous definition that x goes to ∞. In the most general case we can take that x goes to $a \in \mathbb{R} \cup \{-\infty, \infty\}$ in Definition 5. In particular, when we obtain in previous definition that

$$\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} = 0,$$

then we say that f is a little-*o* of g, and written $f \in o(g)$. It is obvious that if $f \in o(g)$, then $f \in O(g)$ is satisfied.

Alternately, we can define big-*O* as follows.

Definition 6. Let f, g: $\mathbb{R} \to \mathbb{R}$. We say that f is $O(g)$ if and only if there are constants $c, N \in \mathbb{R}^+$ such that $|f(x)| \leq c \cdot |g(x)|$, for $x > N$ holds.

Remark 3. The most basic concept concerning the growth of functions is big-*O* notation. The statement that f is big-*O* of g expresses the fact that for large enough x, f will be bounded above by some constant multiple of g. However, Definition 5 and Definition 6 aren't equivalent since there are examples where the Definition 6 holds, but the limit fails to exist in Definition 5. For the functions we will be dealing with, however, this will not happen.

Some important properties of Big-*O* are

1. $f \in O(f)$ (reflexivity of big-*O*),
2. If $f \in O(g)$ and $c \in \mathbb{R} \setminus \{0\}$, then $c \cdot f \in O(g)$,
3. If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$ (transitivity of big-*O*),
4. If $f \in O(h)$ and $g \in O(h)$, then $f + g \in O(h)$,
5. $O(f + g) = \max\{O(f), O(g)\}$,
6. $O(f \cdot g) = O(f) \cdot O(g)$.

The properties 4. and 2. are useful because of the following fact: if a function $f(x)$ is a sum of functions, one of which grows faster than the others, then the faster growing one determines the order of $f(x)$.

Example 2.

If $f(x) = 13 \log x + 4 \log^4 x + 5x + 6x^2 + 2x^3$, then $f(x) \in O(x^3)$.
If $f(x) = 2 \cdot 3^x + 6x^3$, then $f(x) \in O(3^x)$.
If $f(x) = \sqrt[4]{6x^4 + 3x^2 + 4x + 7}$, then $f(x) \in O(x)$.
In general a polynomial is of the order of its highest term.

Example 3. Let $f(x) = \log_2 x$ and $g(x) = \log_9 x$. Is $f \in O(g)$?

Solution. If we recall the identity

$$\log_a b = \frac{\log_c b}{\log_c a},$$

then, we have $f(x) = C \cdot g(x)$, where $C = \log_2 9$, and $f \in O(g)$.

Complexity of an algorithm

When you are analyzing an algorithm or code for its computational complexity using Big-*O* notation, you can ignore the primitive operations that would contribute less-important factors to the run-time. Also, you always take the *worst case* behavior for Big-*O*. In our initial problem, for **vectorMax()** ignore the original two variable initializations, the return statement, the comparison, and the setting of *currentMax* in the loop. Notice that the important part of the function is the fact that the loop conditions will change with the size of the array: for each extra element, there will be one more iteration. This is a linear relationship, and therefore $O(n)$.

Here is a list of classes of functions that are commonly encountered when analyzing algorithms. The slower growing functions are listed first.

Notation	Name
$\boldsymbol{O(1)}$	constant
$\boldsymbol{O(\log n)}$	logarithmic
$\boldsymbol{O(n)}$	linear
$\boldsymbol{O(n \log n)}$	$n \log n$
$\boldsymbol{O(n^2)}$	quadratic
$\boldsymbol{O(n^r)}$	Polynomial ($r > 1$, other than n^2)
$\boldsymbol{O(r^n)}$	Exponential ($r > 1$)

We will show how these functions vary with n. Assume a rate of 1 instruction per *μsec* (micro-second):

Notation	$n = 10$	10^2	10^3	10^4	10^5	10^6
$\boldsymbol{O(1)}$	1 *μsec*	1 *μsec*	1 *μsec*	1 *μsec*	1 *μsec*	1 *μsec*
$\boldsymbol{O(\log n)}$	3 *μsec*	7 *μsec*	10 *μsec*	13 *μsec*	17 *μsec*	20 *μsec*
$\boldsymbol{O(n)}$	10 *μsec*	100 *μsec*	1 *msec*	10 *msec*	100 *msec*	1 *sec*
$\boldsymbol{O(n \log n)}$	33 *μsec*	664 *μsec*	10 *msec*	13.3 *msec*	1.6 *msec*	20 *sec*
$\boldsymbol{O(n^2)}$	100 *μsec*	10 *msec*	1 *sec*	1.7 *min*	16.7 *min*	11.6 *days*
$\boldsymbol{O(n^3)}$	1 *msec*	1 *sec*	16.7 *min*	11.6 *days*	31.7 *year*	31709 *year*
$\boldsymbol{O(2^n)}$	10 *msec*	3e17 *years*				

So algorithms of $O(n^3)$ or $O(2^n)$ are not practical for any more than a few iterations. Quadratic, $O(n^2)$ (common for nested "for" loops!) gets pretty ugly for $n > 1000$.

Now we will give some examples how to determinate computational complexity using Big-*O* notation for iterative and recursive algorithms.

The running time of iterative algorithms is straightforward to compute. Let $f_1(n)$ be the time it takes one iteration of the algorithm to run, and let $f_2(n)$ be the number of iterations. Then the running time of the algorithm is $O(f_1(n) \cdot f_2(n))$.

Example 4.

1.

```
int increment(int n) {
return n + 1;
}
```

This function has one subexpression that takes constant time to execute and executes only once. So it runs in $O(1)$.

2.

```
int factorial(int n) {
for (int i = n; i > 0; i++) {
n *= i;
}
return n;
}
```

This function has a subexpression n *= i that takes constant time to execute, and this subexpression is executed n times. So $f_1(n) = 1,\ f_2(n) = n$, and the function runs in $O(n)$ time.

3.

```
int foo(int n) {
int x;
for (int i = 0; i < n; i++) {
x += i;
}
for (int i = 1; i < n/2; i++) {
x *= i;
}
return x;
}
```

In this case, there are two loops. The first runs in $O(n)$, and the second in $O\left(\frac{n}{2}\right) = O(n)$, so the total running time is in $O(n)$.

4.

```
int bar(int n) {
int x;
for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
        x += 1;
        }
}
return x;
}
```

This function has one subexpression, the inner loop, that executes n times. What is the running time of the subexpression? The subexpression has one subexpression of its own that executes $n - i$ times and is constant. So the running time of the inner loop is in $O(n - i)$. Now the problem with determining the running time of the function is that i varies. But we can make estimates, as long as the estimates are greater than the actual value, so let's assume that the running time of the inner loop is n. Now the inner loop executes n times, so the total running time is in $O(n^2)$.

5.

```
int baz(int k, int n) {
int res = 0;
for (int i = 0; i < k; i++) {
        res += (k - i) * k;
}
for (int i = 0; i < n; i++) {
        res -= (n - i) * i;
}
return res;
}
```

This functions has two loops, the first of which is in $O(k)$ and the second of which is in $O(n)$. We don't know which of the two loops is faster, since it depends on the relative sizes of k and n, so we can only say that the function runs in $O(k + n)$. It is also possible to say that the function runs in $O(max(k, n))$ since we can give an upper bound on the faster loop by assuming it runs in the same amount of time as the slower loop. Note that in general, it is not possible to give the running time of a multiple input function in terms of only one of its inputs.

6.

```
int foobar(int k, int n) {
int res = 0;
for (int i = 0; i < k; i++) {
        for (int j = 0; j < n; j++) {
                res += (k - i) * j;
        }
}
return res;
}
```

The inner loop runs in $O(n)$ time, and the outer loop iterates k times, so the running time of this function is in $O(k \cdot n)$.

Recursive algorithms are somewhat harder to analyze than iterative algorithms. They usually require inductive analysis. We start at the base case and work our way up higher inputs until we see a pattern. One way that helps is to draw a tree of the recursive calls, with each call as a node and an edge between the caller and the callee. We then count how many nodes are in the tree as a function of the input. Then the running time of the algorithm is the number of nodes in the tree times the amount of time each call takes (not including the recursive calls each each call makes).

Example 5.

1.
```
int factorial2(int n) {
if (n == 0) {
return 1;
} else {
return n * factorial2(n - 1);
}
}
```

We draw a tree of the recursive calls in Figure 5. About n recursive calls are made, and each call takes constant time, so the running time of **factorial()** is in $O(n)$.

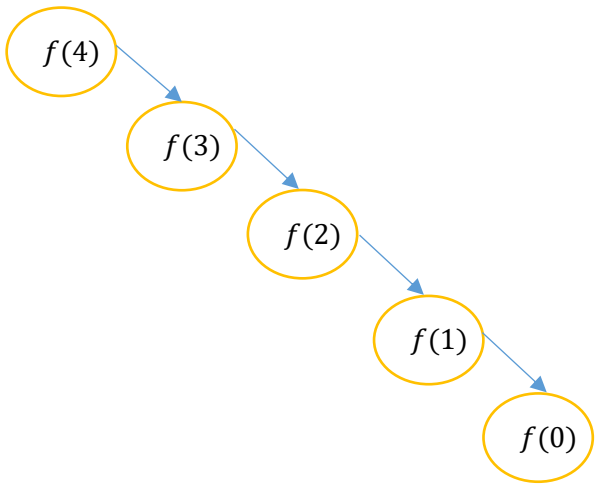


Figure 5. Tree of recursive calls for **factorial(4)**.

2.

```
int fibonacci(int n) {
if (n == 0 || n == 1) {
return 1;
} else {
return fibonacci(n - 1) + fibonacci(n - 2);
}
}
```

Again we draw a tree of the recursive calls in Figure 6. The tree is a nearly complete binary tree, so it has about $2n$ nodes in it. So the running time of this function is in $O(2^n)$.

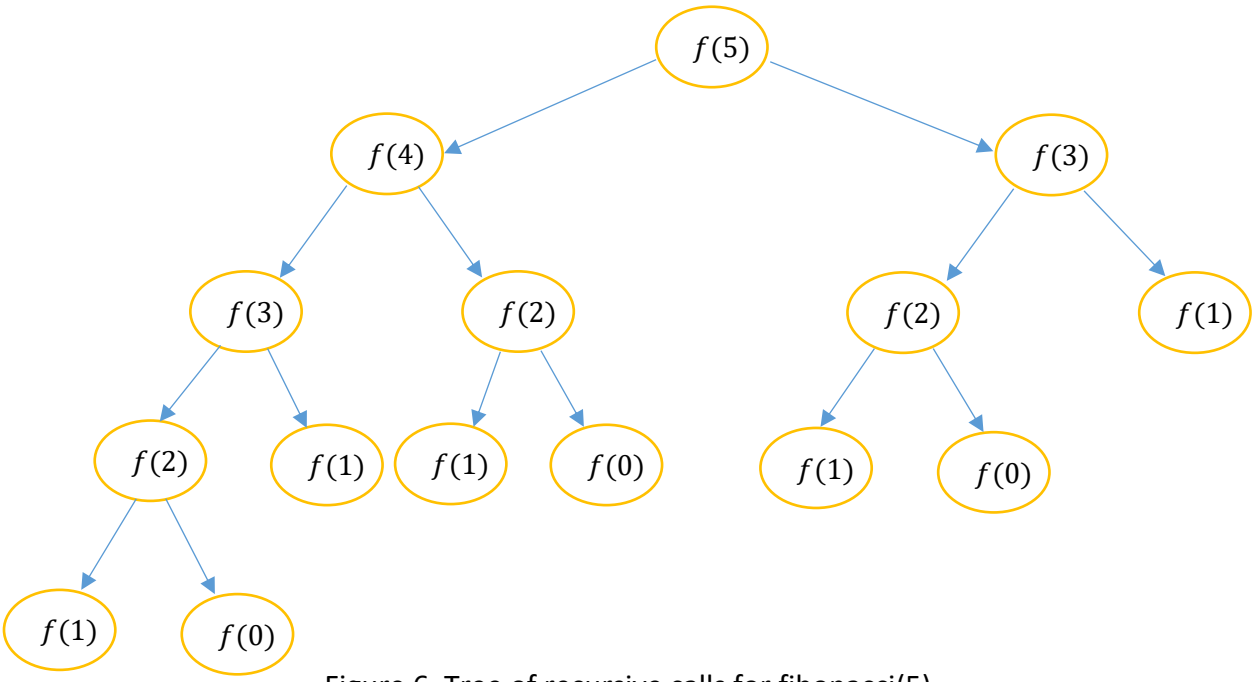


Figure 6. Tree of recursive calls for fibonacci(5).

Finally, we will mention other types of notations that are used in computer science to describe the performance or complexity of an algorithm. Firstly, we will define the Big-Ω notation as follows.

Definition 7. Given a function $g: \mathbb{R} \to \mathbb{R}$

$$\Omega(g) = \left\{ f(x): \lim_{x \to \infty} \frac{|f(x)|}{|g(x)|} > 0 \right\}.$$

$f \in \Omega(g)$ means that f is equal or greater order than g.

The difference between Big-*O* notation and Big-Ω notation is that Big-*O* is used to describe the worst case running time for an algorithm. But, Big-Ω notation, on the other hand, is used to describe the best case running time for a given algorithm.

Is easy to see that $g \in O(f)$ if and only if $f \in \Omega(g)$.

Example 6. For functions $f(x) = 3^x$ and $g(x) = 2^x$ we have

$$\lim_{x\to\infty} \frac{3^x}{2^x} = \infty,$$

so $3^x \in \Omega(2^x)$. On the other hand, we have

$$\lim_{x\to\infty} \frac{2^x}{3^x} = 0,$$

so $2^x \in o(3^x)$, and hence $2^x \in O(3^x)$.

Now, we will define the Big-Θ notation as follows.

Definition 8. Given a function g: $\mathbb{R} \to \mathbb{R}$

$$\Theta(g) = \left\{ f(x) : \lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} = L, 0 < L < \infty \right\}.$$

$f \in \Theta(g)$ means that f and g are roughly the same order.

It is easy to see that $f \in \Theta(g)$ if and only if $f \in O(g)$ and $f \in \Omega(g)$.

Conclusion

- Rather than specifying the exact relation between an algorithm's input and its running time, we only specify how the running time scales as the input grows. For example, if the running time for an algorithm with input n is, $4n^2$ we say that it's running time scales as n^2.
- Also rather than giving the exact relation, we are usually interested in limits on how fast or slow an algorithm is. So we define the following notation:
    1. We say that $f(n)$ is bounded above by $g(n)$ if for all $n > M$ and for some $K > 0$, $K \cdot |g(n)| \geq |f(n)|$. In words, $g(n)$ is an upper bound for $f(n)$ if some positive multiple of $|g(n)|$ is always greater than or equal to $|f(n)|$ after some arbitrary number M. Notice that this definition ignores both constant multiplicative factors and behavior for small inputs.
    2. Similarly, we say that $f(n)$ is bounded below by $g(n)$ if for all $n > M$ and for some $K > 0$, $K \cdot |g(n)| \leq |f(n)|$. In words, $g(n)$ is a lower bound for $f(n)$ if some positive multiple of $|g(n)|$ is always less than or equal to $|f(n)|$ after some arbitrary number M.
    3. We define a set of functions $O(g)$ such that $g(n)$ provides a lower bound for all functions in $O(g)$. In other words, $f(n) \in O(g)$ if $g(n)$ is a lower bound for $f(n)$.
    4. We define a set of functions $\Omega(g)$ such that $g(n)$ provides an upper bound for all functions in $\Omega(g)$. In other words, $f(n) \in \Omega(g)$ if $g(n)$ is a upper bound for $f(n)$.

5. We define a set of functions $\Theta(g)$ such that $g(n)$ provides both an upper bound and a lower bound for all functions in $\Theta(g)$. In other words, $f(n) \in \Theta(g)$ if $g(n)$ is both an upper bound and a lower bound for $f(n)$.

- We can specify the speed of an algorithm by giving functions $g(n)$ and $h(n)$ such that its running time is in $O(g)$ and in $\Omega(h)$. If $g(n) = h(n)$, then its running time is in $\Theta(g)$.

Exercises

1. Assume that each of the expressions below gives the processing time $T(n)$ spent by an algorithm for solving a problem of size n. Select the dominant term(s) having the steepest increase in n and specify the lowest Big-*O* complexity of each algorithm.

Expression	Dominant term	$O(n)$
$5n + 0.001n^3 + 0.025n$		
$500n + 100n^{1.5} + 50n \log_{10} n$		
$0.3n + 5n^{1.5} + 2.5n^{1.75}$		
$n^2 \log_2 n + n(\log_2 n)^2$		
$n \log_2 n + n \log_3 n$		
$3\log_8 n + \log_2 \log_2 \log_2 n$		
$100n + 0.01n^2$		
$0.01n + 100n^2$		
$2n + n^{0.5} + 0.5n^{1.25}$		
$0.01n \log_2 n + n(\log_2 n)^2$		
$100n \log_3 n + n^3 + 100n$		
$0.003\log_4 n + \log_2 \log_2 n$		

2. The statements below show some features of Big-*O* notation for the functions $f = f(n)$ and $g = g(n)$. Determine whether each statement is TRUE or FALSE and correct the formula in the latter case.

Expression	Is it TRUE or FALSE?	If it is FALSE then the correct formula
Rule of sums: $O(f + g) = O(f) + O(g)$		
Rule of products: $O(f \cdot g) = O(f) \cdot O(g)$		

Transitivity: if $g = O(f)$ and $h = O(f)$ then $g = O(h)$		
$5n + 8n^2 + 100n^3 = O(n^4)$		
$5n + 8n^2 + 100n^3 = O(n^2 \log_2 n)$		

3. Algorithms *A* and *B* spend exactly $T_A(n) = 0.1n^2 \log_{10} n$ and $T_A(n) = 2.5n^2$ microseconds, respectively, for a problem of size n. Choose the algorithm, which is better in the Big-*O* sense, and find out a problem size n_0 such that for any larger size $n > n_0$ the chosen algorithm outperforms the other. If your problems are of the size $n \leq 10^9$, which algorithm will you recommend to use?

4. Algorithms *A* and *B* spend exactly $T_A(n) = 5n \log_{10} n$ and $T_B(n) = 25n$ microseconds, respectively, for a problem of size n. Which algorithm is better in the Big-*O* sense? For which problem sizes does it outperform the other?

Solutions

1.

Expression	**Dominant term (s)**	$\boldsymbol{O}(\dots)$
$5n + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n \log_{10} n$	$100n^{1.5}$	$O(n^{1.25})$
$0.3n + 5n^{1.5} + 2.5n^{1.75}$	$2.5n^{1.75}$	$O(n^{1.75})$
$n^2 \log_2 n + n(\log_2 n)^2$	$n^2 \log_2 n$	$O(n^2 \log n)$
$n \log_2 n + n \log_3 n$	$n \log_2 n$, $n \log_3 n$	$O(n \log n)$
$3\log_8 n + \log_2 \log_2 \log_2 n$	$3\log_8 n$	$O(\log n)$
$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$100n^2$	$O(n^2)$
$2n + n^{0.5} + 0.5n^{1.25}$	$0.5n^{1.25}$	$O(n^{1.25})$
$0.01n \log_2 n + n(\log_2 n)^2$	$n(\log_2 n)^2$	$O(n(\log n)^2)$
$100n \log_3 n + n^3 + 100n$	n^3	$O(n^3)$
$0.003\log_4 n + \log_2 \log_2 n$	$0.003\log_4 n$	$O(\log n)$

2.

Expression	Is it TRUE or FALSE?	If it is FALSE then the correct formula
Rule of sums: $O(f+g) = O(f) + O(g)$	FALSE	$O(f+g) = \max\{O(f), O(g)\}$
Rule of products: $O(f \cdot g) = O(f) \cdot O(g)$	TRUE	
Transitivity: if $g = O(f)$ and $h = O(f)$ then $g = O(h)$	FALSE	if $g = O(f)$ and $f = O(h)$ then $g = O(h)$
$5n + 8n^2 + 100n^3 = O(n^4)$	TRUE	
$5n + 8n^2 + 100n^3 = O(n^2 \log_2 n)$	FALSE	$5n + 8n^2 + 100n^3 = O(n^3)$

3. In the Big-*O* sense, the algorithm *B* is better. It outperforms the algorithm A when $T_B(n) \leq T_A(n)$, that is, when $2.5n^2 \leq 0.1n^2 \log_{10} n$. This inequality reduces to $\log_{10} n \geq 25$, or $n \geq n_0 = 10^{25}$. If $n \leq 10^9$, the algorithm of choice is *A*.

4. In the Big-*O* sense, the algorithm *B* is better. It outperforms the algorithm A if $T_B(n) \leq T_A(n)$, that is, if $25n \leq 5n \log_{10} n$, or $\log_{10} n \geq 5$, or $n \geq 100000$.