DESIGN & ANALYSIS OF ALGORITHMS (10CS43)

UNIT-VI

LIMITATIONS OF ALGORITHMIC POWER AND COPING WITH THEM

Topics:

- ✔ **Lower-Bound Arguments**
- ✔ **Decision Trees**
- ✔ **P, NP, and NP-Complete Problems**
- ✔ **Challenges of Numerical Algorithms.**

I. Lower-Bound Arguments

One can look at the efficiency of an algorithm two ways; by establish its

1. *asymptotic efficiency class* and other approach is to
2. *ask how efficient a particular algorithm* is with respect to other algorithms for the same problem.

To determine the efficiency of an algorithm with respect to other algorithms for the same problem, it is desirable to know the best possible efficient algorithm solving that problem may have. Knowing such a ***lower bound*** can tell us how much improvement we can hope to achieve in for a better algorithm.

The several methods for establishing lower bounds are

a) **Trivial Lower Bounds**
b) **Information-Theoretic Arguments**
c) **Adversary Arguments**
d) **Problem Reduction**

a. Trivial Lower Bounds

The simplest method of obtaining a lower-bound class is based on *counting the number of items in the problem's input that must be processed and the number of output items that need to be produced.* Since any algorithm must at least "read" all the items it needs to process and "write" all its outputs, such a count yields a ***trivial lower bound***.

For example, any algorithm for **generating all permutations** of n distinct items in ***(n!)*** because the size of the output is ***n!*** And this bound is tight because good algorithms for generating permutations spend a constant time on each of them except the initial one.

As another example, consider the **problem of evaluating a polynomial** of degree *n*

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$$

At a given point *x*, given its coefficients $a_n, a_{n-1}, \ldots, a_0$. It is easy to see that all the coefficients have to be processed by any polynomial-evaluation algorithm. This means that any such algorithm must be in Ω *(n)*. This lower bound is tight because both the right-to-left evaluation algorithm is linear.

A trivial lower bound for **computing the product of two *n* × *n* matrices** is $\Omega(n^2)$ because any such algorithm has to process $2n^2$ elements in the input matrices and generate n^2 elements of the product.

For example, the trivial bound for the **traveling salesman problem** is $\Omega(n^2)$, because its input is $n(n-1)/2$ intercity distances and its output is a list of $n+1$ cities making up an optimal tour. But this bound is all but useless because there is no known algorithm with the running time being a polynomial function of any degree.

Another example, **searching for an element** of a given value in a sorted array does not require processing all its elements.

Consider the problem of **determining connectivity of an undirected graph** defined by its adjacency matrix. It is reasonable to expect that any such algorithm would have to check the existence of each of the $n(n-1)/2$ potential edges, but the proof of this fact is not trivial.

b. Information-Theoretic Arguments

The information-theoretical approach establishes a lower bound based on the amount of information it has to produce. Consider, as an example, the well-known game of deducing a positive integer between 1 and *n* selected by somebody by asking that person questions with yes/no answers. The amount of uncertainty that any algorithm solving this problem has to resolve can be measured by $\log_2 n$, the number of bits needed to specify a particular number among the *n* possibilities. We can think of each question as yielding at most 1 bit of information about the algorithm's output, i.e., the selected number. Consequently, any such algorithm will need atleast $\log_2 n$ such steps before it can determine its output in the worst case. The approach is called the ***information-theoretic argument*** because of its connection to information theory. It has proved to be quite useful for finding the so-called ***information-theoretic lower bounds*** for many problems involving comparisons, including sorting and searching.

c. Adversary(Challenger) Arguments

Consider the example; consider the problem of merging two sorted lists of size n

$$a_1 < a_2 < \ldots < a_n \text{ and } b_1 < b_2 < \ldots < b_n$$

into a single sorted list of size $2n$.

Merging is done by repeatedly comparing the first elements in the remaining lists and outputting the smaller among them. The number of key comparisons in the worst case for this algorithm for merging is $2n - 1$.

Knuth quotes the following adversary method for proving that $2n - 1$ is a lower bound on the number of key comparisons made by any comparison-based algorithm for this problem. The adversary will employ the following rule: reply true to the comparison $ai<bj$ if and only if $i < j$. This will force any correct merging algorithm to produce the only combined list consistent with this rule:

$$b_1 < a_1 < b_2 < a_2 < \ldots < b_n < a_n.$$

To produce this combined list, any correct algorithm will have to explicitly compare $2n - 1$ adjacent pairs of its elements, i.e., b_1 to a_1, a_1 to b_2, and so on. If one of these comparisons has not been made, e.g., a_1 has not been compared to b_2, we can transpose these keys to get $b_1 < b_2 < a_1 < a_2 < \ldots < b_n < a_n$, which is consistent with all the comparisons made but cannot be distinguished from the correct configuration given above. Hence, $2n - 1$ is, indeed, a lower bound for the number of key comparisons needed for any merging algorithm.

d. Problem Reduction

The idea is getting an algorithm for problem ***P*** by reducing it to another problem solvable with a known algorithm. A similar reduction idea can be used for finding a lower bound. To show that problem ***P*** is at least as hard as another problem ***Q*** with a known lower bound, we need to reduce Q to P. In other words, we should show that an arbitrary instance of problem Q can be transformed to an instance of problem P, so any algorithm solving P would solve Q as well. Then a lower bound for Q will be a lower bound for P

2. Decision Trees

Many important algorithms, especially those for sorting and searching, work by comparing items of their inputs. We can study the performance of such algorithms with a device called a ***decision tree***.

As an example, Figure presents a decision tree of an algorithm for finding a **minimum of three numbers.**

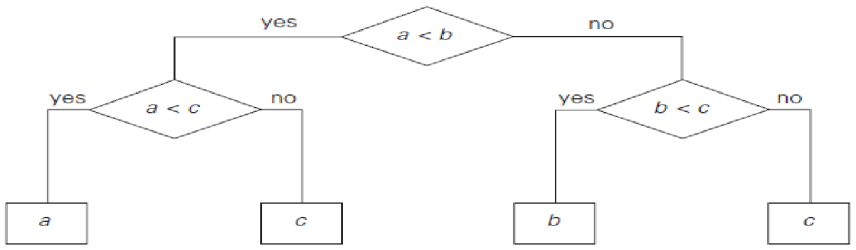


FIGURE 11.1 Decision tree for finding a minimum of three numbers.

Each internal node of a binary decision tree represents a key comparison indicated in the node, e.g., $k < k'$. The node's left subtree contains the information about subsequent comparisons made if $k < k'$, and its right subtree does the same for the case of $k > k'$. Each leaf represents a possible outcome of the algorithm's run on some input of size n. Note that the number of leaves can be greater than the number of outcomes because, for some algorithms, the same outcome can be arrived at through a different chain of comparisons. An important point is that the number of leaves must be at least as large as the number of possible outcomes. The algorithm's work on a particular input of size n can be traced by a path from the root to a leaf in its decision tree, and the number of comparisons made by the algorithm on such a run is equal to the length of this path. Hence, the number of comparisons in the worst case is equal to the height of the algorithm's decision tree.

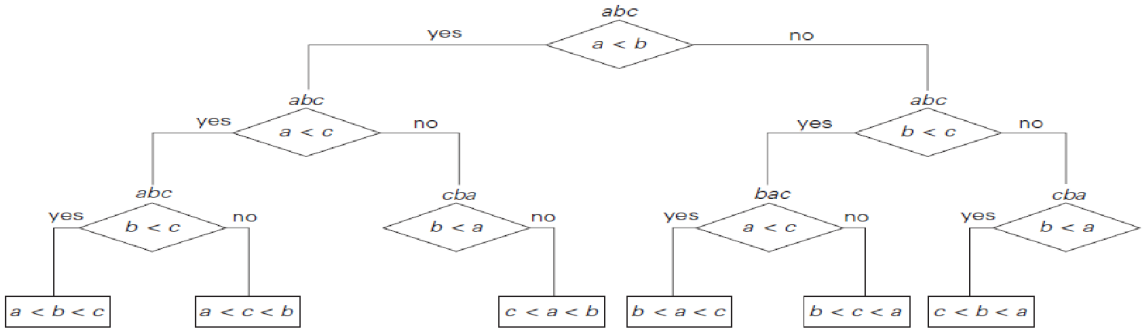
The central idea behind this model lies in the observation that a tree with a given number of leaves, which is dictated by the number of possible outcomes, has to be tall enough to have that many leaves. Specifically, it is not difficult to prove that for any binary tree with l leaves and height h,

$$h \geq \lceil \log_2 l \rceil.$$

Decision Trees for Sorting

Most sorting algorithms are comparison based, i.e., they work by comparing elements in a list to be sorted. By studying properties of decision trees for such algorithms we can derive important lower bounds on their time efficiencies.

We can interpret an outcome of a sorting algorithm as finding a permutation of the element indices of an input list that puts the list's elements in ascending order using **selection sort**. Consider, as an example, a three-element list *a, b, c* of orderable items such as real numbers or strings. For the outcome $a < c < b$ obtained by sorting this list, the permutation in question is 1, 3, 2. In general, the number of possible outcomes for sorting an arbitrary *n*-element list is equal to *n*!.



The height of a binary decision tree for any comparison based sorting algorithm and hence the worst-case number of comparisons made by such an algorithm cannot be less than $\lceil \log_2 n! \rceil$:

$$C_{worst}(n) \geq \lceil \log_2 n! \rceil.$$

We can also use decision trees for analysing the average-case efficiencies of comparison-based sorting algorithms. We can compute the average number of comparisons for a particular algorithm as the average depth of its decision tree's leaves, i.e., as the average path length from the root to the leaves. For example, for

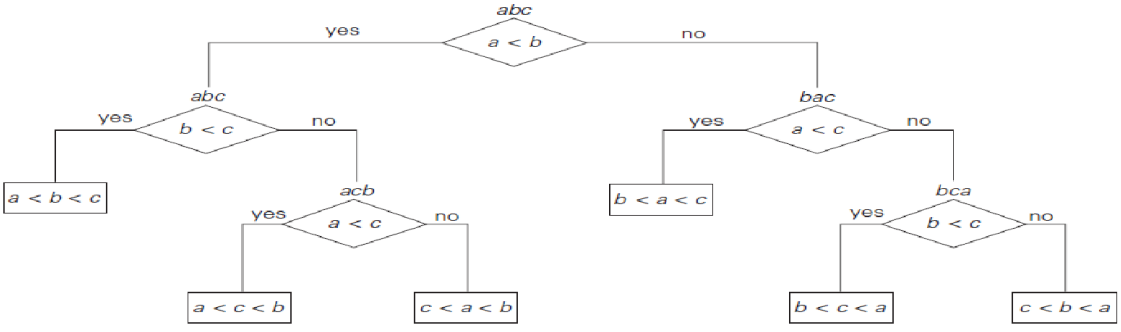


FIGURE 11.3 Decision tree for the three-element insertion sort.

the three-element insertion sort whose decision tree is given in Figure, this number is (2 + 3 + 3 + 2 + 3 + 3)/6 = 2(2/3). Under the standard assumption that all $n!$ outcomes of sorting are equally likely, the following lower bound on the average number of comparisons made by any comparison-based algorithm in sorting an n-element list has been proved: $C_{avg}(n) \geq \log_2 n!$.

Decision Trees for Searching a Sorted Array

In this section, we shall see how decision trees can be used for establishing lowerbounds on the number of key comparisons in searching a sorted array of n keys: $A[0]<A[1]<\ldots<A[n-1]$. The principal algorithm for this problem is binary search. The number of comparisons made by binary search in the worst case, $C^{bs}_{worst}(n)$, is given by the formula

$$C^{bs}_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil.$$

We can use decision trees to determine whether this is the smallest possible number of comparisons. Since we are dealing here with three-way comparisons in which search key K is compared with some element $A[i]$ to see whether $K<A[i], K = A[i]$, or $K>A[i]$,it is natural to try using ternary decision trees. Figure 11.4 presents such a tree for the case of $n = 4$.

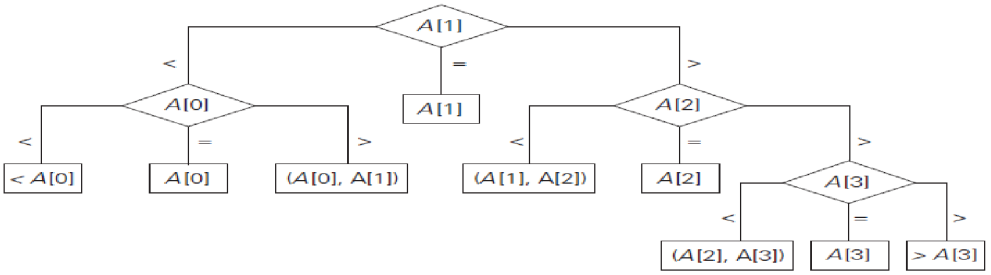


FIGURE 11.4 Ternary decision tree for binary search in a four-element array.

The internal nodes of that tree indicate the array's elements being compared with the search key. The leaves indicate either a matching element in the case of a successful search or a found interval that the search key belongs to in the case of an unsuccessful search. We can represent any algorithm for searching a sorted array by three-way comparisons with a ternary decision tree similar to that in Figure 11.4. For an array of n elements, all such decision trees will have $2n + 1$ leaves (n for successful searches and $n + 1$ for unsuccessful ones). Since the minimum height h of a ternary tree with l leaves is _log3 l_, we get the following lower bound on the number of worst-case comparisons: $C_{worst}(n) \geq \lceil \log_3 (2n+1) \rceil$.

This lower bound is smaller that $\lceil \log_2 (n+1) \rceil$, the number of worst case comparisons for binary search, at least for large values of n. To obtain a better lower bound, we should

consider binary rather than ternary decision trees, such as the one in Figure 11.5. Internal nodes in such a tree correspond to the same three way comparisons as before, but they also serve as terminal nodes for successful searches. Leaves therefore represent only unsuccessful searches, and there are $n + 1$ of them for searching an n-element array.

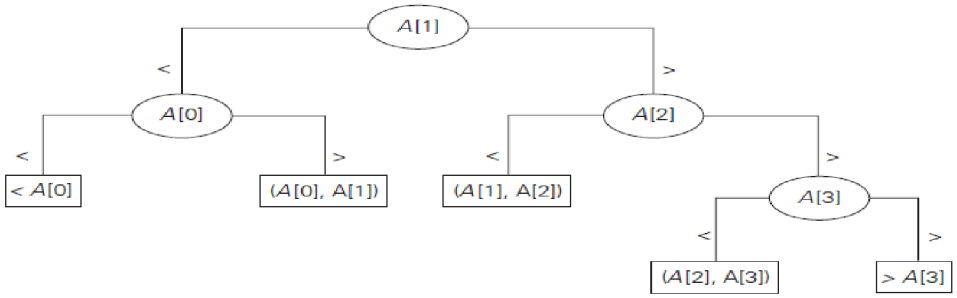


FIGURE 11.5 Binary decision tree for binary search in a four-element array.

As comparison of the decision trees in Figures 11.4 and 11.5 illustrates, the binary decision tree is simply the ternary decision tree with all the middle subtrees eliminated. Applying inequality (11.1) to such binary decision trees immediately yields

$$C_{worst}(n) \geq \lceil \log_2(n+1) \rceil.$$

This inequality closes the gap between the lower bound and the number of worst case comparisons made by binary search, which is also $\lceil \log_2(n+1) \rceil$. A much more sophisticated analysis shows that under the standard assumptions about searches, binary search makes the smallest number of comparisons on the average, as well. The average number of comparisons made by this algorithm turns out to be about log2 n and log2$(n + 1)$ for successful and unsuccessful searches, respectively.

3 *P*, *NP*, and *NP*-Complete Problems

In the study of the computational complexity of problems, the first concern of both computer scientists and computing professionals is whether a given problem can be solved in polynomial time by some algorithm.

DEFINITION : We say that an algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to $O(p(n))$ where $p(n)$ is a polynomial of the problem's input size n. Problems that can be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.

There are several reasons for drawing the intract ability line in this way.

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

First, the entries of Table imply that we cannot solve arbitrary instances of intractable problems in a reasonable amount of time unless such instances are very small.

Second, even if there might be a huge difference between the running times in *O (p (n))* for polynomials of drastically different degrees, there are very few useful polynomial-time algorithms with the degree of a polynomial higher than three. Polynomials that bound running times of algorithms do not usually have extremely large coefficients.

Third, polynomial functions possess many convenient properties; in particular, both the sum and composition of two polynomials are always polynomials too.

Fourth, the choice of this class has led to a development of an extensive theory called ***computational complexity***, which seeks to classify problems according to their inherent difficulty.

P and NP Problems

DEFINITION Class *P* is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called ***polynomial***.

There are many important problems, however, for which no polynomial-time algorithm has been found, nor has the impossibility of such an algorithm been proved. Some of the best known problems that fall into this category:

1. ***Hamiltonian circuit problem*** Determine whether a given graph has a Hamiltonian circuit—a path that starts and ends at the same vertex and passes through all the other vertices exactly once.
2. ***Traveling salesman problem*** Find the shortest tour through n cities with known positive integer distances between them.
3. ***Knapsack problem*** Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.

4. ***Partition problem*** Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.
5. ***Bin-packing problem*** Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.
6. ***Graph-colouring problem*** For a given graph, find its chromatic number, which is the smallest number of colours that need to be assigned to the graph's vertices so that no two adjacent vertices areassigned the same colour.
7. ***Integer linear programming problem*** Find the maximum value of a linear function.

DEFINITION

A ***nondeterministic algorithm*** is a two-stage procedure that takes as its input an instance *I* of a decision problem and does the following. Nondeterministic ("guessing") stage: An arbitrary string *S* is generated that can be thought of as a candidate solution to the given instance *I*(but may be complete gibberish as well). Deterministic ("verification") stage: A deterministic algorithm takes both *I* and *S* as its input and outputs yes if *S* represents a solution to instance *I.* (If *S* is not a solution to instance *I,* the algorithm either returns no or is all owed not to halt at all.)

A nondeterministic algorithm solves a decision problem if and only if for every yes instance of the problem it returns yes on some execution. Finally, a nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time efficiency of its verification stage is Polynomial. Now we can define the class of *NP* problems.

DEFINITION

Class *NP* is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called ***nondeterministicPolynomial***.

Most decision problems are in *NP*. First of all, this class includes all the problems in *P*:

$$P \subseteq NP.$$

This is true because, if a problem is in *P*, we can use the deterministic polynomial time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simply ignores string *S* generated in its nondeterministic ("guessing") stage.

***NP*-Complete Problems**

Informally, an *NP*-complete problem is a problem in *NP* that is as difficult as any other problem in this class because, by definition, any other problem in *NP* can be reduced to it in polynomial time (shown symbolically in Figure.

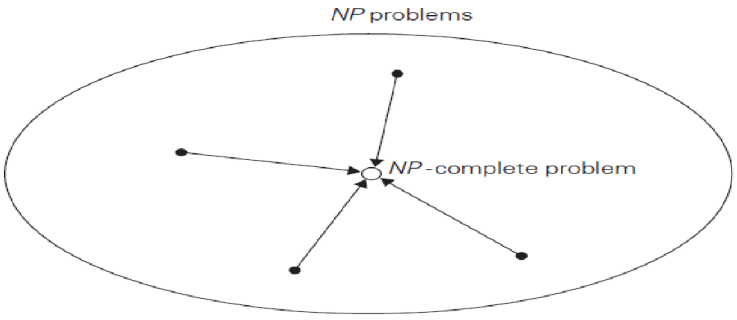


FIGURE 11.6 Notion of an *NP*-complete problem. Polynomial-time reductions of *NP* problems to an *NP*-complete problem are shown by arrows.

Here are more formal definitions of these concepts.

DEFINITION

A decision problem *D*1 is said to be ***polynomially reducible*** to a decision problem *D*2, if there exists a function *t* that transforms instances of *D*1to instances of *D*2 such that:

1. i*t* maps all yes instances of *D*1 to yes instances of *D*2 and all no instances of *D*1to no instances of *D*2

2. i*t* is computable by a polynomial time algorithm This definition immediately implies that if a problem *D*1 is polynomially reducible to some problem*D*2 that can be solved in polynomial time, then problem*D*1 can also be solved in polynomial time (why?).

NP-complete

DEFINITION

A decision problem *D* is said to be ***NP-complete if***:

1. It belongs to class *NP*

2. Every problem in *NP* is polynomially reducible to *D*

The definition of *NP*-completeness immediately implies that if there exists a deterministic polynomial-time algorithm for just one *NP* complete problem, then every problem in *NP* can be solved in polynomial time by a deterministic algorithm, and hence *P* = *NP*. In other words, finding a polynomial-time algorithm

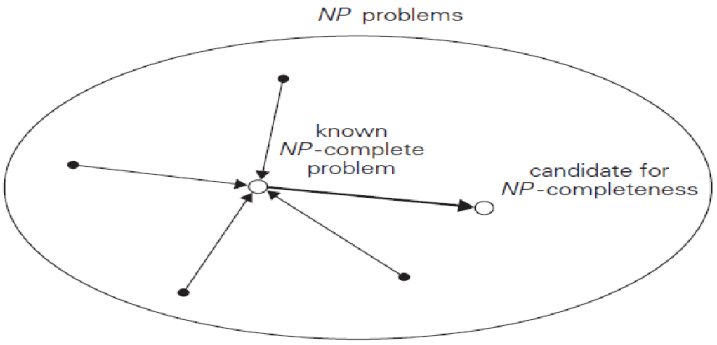


FIGURE 11.7 Proving *NP*-completeness by reduction.

for one *NP*-complete problem would mean that there is no qualitative difference between the complexity of checking a proposed solution and finding it in polynomial time for the vast majority of decision problems of all kinds. Such implications make most computer scientists believe that *P* _= *NP,* although nobody has been successful so far in finding a mathematical proof of this intriguing conjecture.

4 Challenges of Numerical Algorithms

Numerical analysis is described as the branch of computer science concerned with algorithms for solving mathematical problems.

The **first major** problem is that most numerical analysis problems cannot be solved exactly. They have to be solved approximately, and this is usually done by replacing an infinite object by a finite approximation. The errors of such approximations are called ***truncation errors***. One of the major tasks in numerical analysis is to estimate the magnitudes of truncation errors. This is typically done by using calculus tools, from elementary to quite advanced.

The **second type** of errors, called ***round-off errors***, is caused by the limited accuracy with which we can represent real numbers in a digital computer. These errors arise not only for all irrational numbers but for many rational numbers as well. In the great majority of situations, real numbers are represented as floating-point numbers

$$\pm .d_1 d_2 \ldots d_p \cdot B^E,$$

where *B* is the number base, *d*1*, d*2*, . . . , dp* are digits representing together the fractional part of the number and called its ***mantissa***; and *E* is an integer ***exponent*** with the range of values approximately symmetric about 0.The accuracy of the floating-point representation depends on the number of ***significant digits*** *p* . Most computers permit two or even three levels of precision: ***single precision***, ***double precision***, and ***extended precision***. Using higher precision arithmetic slows computations

but may help to overcome some of the problems caused by round-off errors. Higher precision may need to be used only for a particular step of the algorithm in question.

As with an approximation of any kind, it is important to distinguish between the ***absolute error*** and the ***relative error*** of representing a number a^* by its approximation a:

$$\text{absolute error} = |\alpha - \alpha^*|,$$
$$\text{relative error} = \frac{|\alpha - \alpha^*|}{|\alpha^*|}.$$

Very large and very small numbers cannot be represented in floating point arithmetic because of the phenomena called ***overflow*** and ***underflow***, respectively. An overflow happens when an arithmetic operation yields a result outside the range of the computer's floating point numbers. Typical examples of overflow arise from the multiplication of large numbers or division by a very small number. Underflow occurs when the result of an operation is a nonzero fraction of such a small magnitude that it cannot be represented as a nonzero floating-point number.

It is important to remember that, in addition to inaccurate representation of numbers, the arithmetic operations performed in a computer are not always exact, either. In particular, subtracting two nearly equal floating-point numbers may cause a large increase in relative error. This phenomenon is called ***subtractive cancellation***.