Computational complexity
— T.04 —

Summary:

- Algorithms
- Abstract data types
- Computational complexity
- Algorithm analysis
- Asymptotic notation
- Classes of problems
- Useful formulas
- Least squares fit
- A first example
- More examples
- Computational challenge
- Exercises

Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **The Algorithm Design Manual**, chapter 2, Steven S. Skiena, second edition, Springer, 2008.
- **Introduction to Algorithms**, chapters 1 and 3, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Análise da Complexidade de Algoritmos**, chapter 1, António Adrego da Rocha, FCA.

An algorithm is a set of finitely many rules for manipulating a finite amount of data in order to produce a result in a finite number of steps.
— The Princeton Companion to Mathematics, section II.4 (by Jean-Luc Chabert)

Algorithms (part 1, definition)

An algorithm is a detailed description of a method to solve a given problem. To properly specify the problem it is necessary to specify its input, which encodes the information necessary to uniquely identify each instance of the problem, and its output, which encodes the solution to the problem. It is also necessary to specify the properties that the output must have in order for it to be a solution to the problem for the given input (in other words, we must also specify what the problem actually is). An algorithm is a finite sequence of instructions that explain, in painstaking detail, how to produce a valid output (a solution to the problem) given a valid input. These instructions can be given in a natural language, such as English, or in a form closer to how an algorithm may be actually implemented in a given programming language (pseudocode).

An algorithm:

- must be **correct**, i.e., it must produce a valid output for **all** possible valid inputs;
- must be **unambiguous**, i.e., its steps must be precisely defined;
- must be **finite**, i.e., it must produce a valid output in a finite number of steps for **all** possible valid inputs (we are not interested in "algorithms" that may take forever to produce a valid output);
- should be **efficient**, i.e., it should do the job as quickly as possible;
- can be **deterministic**, if the sequence of steps that produces the output depends only on the input, or it
- can be **randomized**, if the sequence of steps that produces the output depends on the input and on random choices.

Algorithms (part 2, example)

The following algorithm is a possible way to sort a sequence of numbers:

Algorithm: Generic exchange sort.

Input: a sequence of n numbers $a_1, a_2, \ldots, a_n$, with $n > 0$.

Output: a permutation (reordering) $b_1, b_2, \ldots, b_n$ of the input sequence such that $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$.

Steps:

1. [Copy input.] For $i = 1, 2, \ldots, n$, set b_i to a_i.
2. [Deal with a special case.] If $n = 1$ terminate the algorithm.
3. [Are we done?] Find a pair of indices (i, j) such that $i < j$ and $b_i > b_j$. Terminate the algorithm if such a pair does not exist.
4. [Exchange.] Exchange b_i with b_j. Return to step 3.

Is the algorithm correct? Yes. For $n > 1$ the algorithm can only terminate when the current b sequence is sorted in non-decreasing order.

Is the algorithm unambiguous? No, because step 3 does not specify how the pair of indices (i, j) is to be found. That is a sub-problem that also has to be fully specified. However, no matter how this is done, the algorithm is correct.

Is the algorithm finite? Yes. Since the maximum possible number of exchanges is $n(n-1)/2$ — that is the number of different pairs (i, j) with $i < j$ — sooner or latter the search in step 3 to find an acceptable pair (i, j) will fail.

Is the algorithm efficient? That depends on how the pair of indices is found in step 3. (If that is done from scratch every time, it will be inefficient.)

Abstract data types

Most algorithms need to organize the data they work with in certain ways. So, all modern programming languages allow the programmer to define new data types that are not predefined in the language. More often than not each particular kind of data used by an algorithm can be stored in more than one way but is used in essentially the same way. In those cases the programmer should choose the more efficient storage organization for each kind of data. What constitutes the best storage organization can change during the development of the program. For example, at first it might be thought that the space required to store the data is more important that the time it takes to query or modify it. Later on it may turn out that it is the other way around. So, a good programmer will pay considerable attention to the operations (transformations, queries) that the algorithm needs to perform on its data, and will define data types not by the specific way they store their information but by what operations are allowed to be performed on the information that is supposed to be stored in each one of them. He/she will design abstract data types.

An abstract data type is a data type that exposes to the rest of the program the ways the data it stores can be queried or modified (the interface), without exposing to the rest of the program its internal workings (the implementation), be it how the data is actually stored or how the queries/modifications are actually performed. This will make the program more modular, because as long as the interface of an abstract data type is not changed, changes in its implementation will not affect how the rest of the program is coded.

A proper specification of the interface defines not only the names and arguments of the operations that can be performed on instances of the data type (that has to be placed in the source code) but also what their side effects are (that should be placed in the source code in the form of comments). Some programming languages provide facilities (assertions, and pre- and post-conditions) that help ensure that the interface of a data type is used correctly.

Computational complexity (part 1, the RAM model of computation)

To be able to compare the efficiency of different algorithms one needs a model of computation. A model of computation quantifies how much work is needed to perform an elementary task, such as performing an arithmetic operation or a memory access.

The RAM model of computation is one of the simplest we can use. It is based on the notion of a Random Access Machine, abbreviated as RAM. Under this model of computation

- an elementary arithmetic operation, such as +, − and the like, a comparison, and an assignment, of numbers with a given fixed number of bits, uses one time step,
- the operation of calling a subroutine (just the call, not the actual work done by the subroutine), of evaluating a numerical transcendental function such as `sin(x)`, and of following a conditional branch, also uses one time step,
- loops, and subroutines, have to be broken down into their individual constituents,
- a memory access, be it a read or a write access, also uses one time step, and
- the memory space used to store a number with a given fixed number of bits uses one unit of space.

This is, of course, a very simplistic model of what happens on a true processor. For example, on a modern processor a division takes much more time than an addition. It is nonetheless a useful model, because in the worst case it deviates (above and below) from what happens on a modern processor by a constant factor.

The computational complexity of an algorithm gives the number of time steps, and the number of units of space, required by the algorithm to solve a given problem. It is a function of the size of the algorithm's input. The size of the input is usually either the number of its data items (say, the number of elements of an array), or one of the inputs itself (say, the exponent in an exponentiation algorithm).

Computational complexity (part 2, a simple example)

Consider the following very simple algorithm:

Algorithm: Mean.
Input: a sequence of n numbers $a_1, a_2, \ldots, a_n$, with $n > 0$.
Output: the arithmetic mean $\mu = \frac{1}{n}\sum_{i=1}^{n} a_i$
Steps:

1. [Initialization.] Set s to a_1.
2. [Sum.] For $i = 2, 3, \ldots, n$, add a_i to s.
3. [Return.] Terminate the algorithm, outputting s/n as the value of μ.

To determine the computational complexity of this algorithm we need to count the number of elementary operations it performs. Step 1 requires one time step to read a_1 (it is read directly into s, so no assignment is needed). For each value of i, step 2 requires five time steps: one to read a_i, another one to add it to s, and another three to increment i, to compare it with n, and to perform a conditional jump according to the result of the comparison. Step 3 requires two time steps, one to perform the division and another to terminate the algorithm (we consider it to be a subroutine return, so we also count it). Since there are $n - 1$ values in step 2, the total number of time steps used by the algorithm is $T(n) = 1 + 5(n - 1) + 2 = 5n - 2$.

Doing such a detailed analysis is usually not necessary. All that one is usually interested in is knowing how fast $T(n)$ grows when n grows. For large n, in this case $T(n)/n$ is almost constant (linear complexity). Knowing that is usually much more important than knowing that $T(n) \approx 5n$; knowledge of the growth constant, 5 in this case, although useful, is in most cases an overkill. (After all, the true constant will depend on the processor where the algorithm will be run and on the optimizations of the code made by the compiler.)

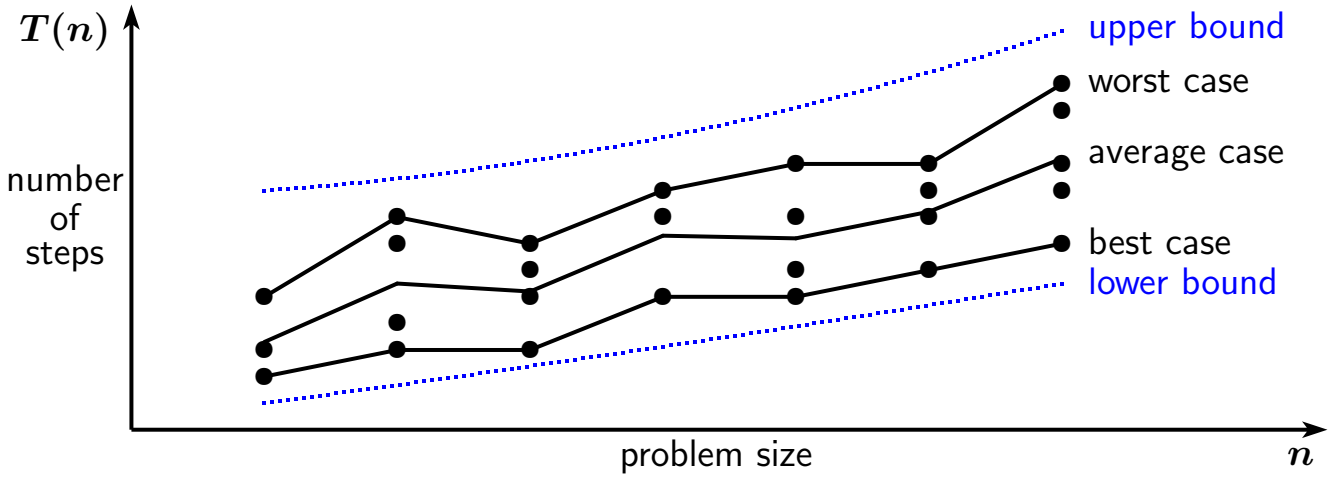
Algorithm analysis (part 1, formal and empirical analysis)

In the last slide we did a formal complexity analysis of a simple algorithm. We have taken the trouble to account for every elementary operation performed by the algorithm. That was easy to do because the algorithm was very easy. For more complex algorithms, specially for those with a flow of execution that depends on the input, that is hard to do, even if one takes a probabilistic approach to the problem. (In a probabilistic approach every conditional branch that depends on the input data has a specific probability of being taken. Assigning these probabilities usually requires a great deal of mathematical sophistication on the part of the person doing the formal complexity analysis.)

When a formal complexity analysis it too difficult to perform, or when we are just too lazy to do it (or when we do not know enough to do it), one can do an empirical complexity analysis. An empirical analysis is based on experimentation. One implements the algorithm using a suitable programming language, and one either adds code to count the number of operations done by the program (this is called instrumenting the program), or one just measures its execution time. In both cases, one has to select a good set of typical inputs of various sizes. The experimental values of $T(n)$ measured in this way can then be plotted as a function of n to see how they grow. With luck (or knowledge), it will be possible to find out a mathematical formula that fits the experimental observations reasonably well.

Algorithm analysis (part 2, worst, best, and average cases)

The worst case time of an algorithm is the function defined by the maximum number of time steps used by the algorithm to deal with any valid input of size n. The best case and average case times are defined in the same way. (Best, worst, and average spaces can also be defined.) As shown in the following figure, these functions may on occasion decrease when the problem size increases.



Best and worst case times are usually easier to determine than the average case time. Furthermore, exact best and worst times are usually more difficult to determine, and to use, than smooth bounds of these functions (in blue in the figure). Obviously, we need a lower bound for the best case and an upper bound for the worst case.

We are usually interested in knowing how fast the lower and upper bounds grow when the size of the problem increases. For example, $T_1(n) = 3n^2$ and $T_2(n) = 10n \log n$ grow at clearly distinct rates, while $T_3(n) = 4n^2$ and $T_4(n) = 3n^2 + 10n$ do not. Mathematicians have a way to express succinctly these differences: asymptotic notation. Using asymptotic notation we talk about the best, average, and worst time **complexity** of an algorithm.

Asymptotic notation (part 1, definitions)

Asymptotic notation allows us to hide irrelevant details about how fast a function grows. For example, when n is a huge number it is overkill to know exactly that $T_1(n) = 2n^2+3000n+5$ and that $T_2(n) = 10n^2+100n-23$. For huge numbers all that matters is that $T_1(n)$ is approximately $2n^2$ and that $T_2(N)$ is approximately $10n^2$, so that $T_2(n)$ will be about 5 times larger than $T_1(n)$. In asymptotic notation the only detail that is kept about $T_1(n)$ and $T_2(n)$ is that they grow like a square function; even the constant factor that multiplies n^2 is hidden behind the mathematical notation.

Asymptotic notation comes in one of several forms (all functions and constants are assumed here to be positive):

- [**Big Oh notation**] The Big Oh notation is useful to deal with **upper bounds**. The notation
  $$f(n) = O(g(n))$$
  means that there exists an n_0 and a constant C such that, for all $n \geqslant n_0$, $f(n) \leqslant Cg(n)$.
- [**Big Omega notation**] The Big Omega notation is useful to deal with **lower bounds**. The notation
  $$f(n) = \Omega(g(n))$$
  means that there exists an n_0 and a constant C such that, for all $n \geqslant n_0$, $f(n) \geqslant Cg(n)$.
- [**Big Theta notation**] The Big Theta notation is useful to deal with **upper and lower bounds** that have the **same form**. The notation
  $$f(n) = \Theta(g(n))$$
  means that there exists an n_0 and two constants C_1 and C_2 such that for all $n \geqslant n_0$ one has $C_1g(n) \leqslant f(n) \leqslant C_2g(n)$.
- [**small oh notation**] The notation
  $$f(n) = o(g(n))$$
  means that $f(n)$ grows slower than $g(n)$, i.e., that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Asymptotic notation (part 2, properties)

The notation introduced in the previous slide can be used in an expression. For example, $f(n) = n^2 + O(n)$ means that $f(n)$ deviates from n^2 by a quantity that is $O(n)$, that is, whose absolute value is bounded by a multiple of n. (The definitions introduced before can be extended to general functions by using absolute values.) We may, for example, say that $O(2n^2 - \log n) = O(3n^2 + 100n - 23)$, because the absolute value of both functions can be bounded by a quadratic, i.e., both are $O(n^2)$.

What can we say about $O\big(f(n) + g(n)\big)$? There are three cases to consider:

- $f(n) = \Theta\big(g(n)\big)$, which implies that $g(n) = \Theta\big(f(n)\big)$. In this case $O\big(f(n) + g(n)\big) = O\big(f(n)\big)$.
- $f(n) = o\big(g(n)\big)$. In this case $O\big(f(n) + g(n)\big) = O\big(g(n)\big)$.
- $g(n) = o\big(f(n)\big)$. In this case $O\big(f(n) + g(n)\big) = O\big(f(n)\big)$.

In all cases the function that grows faster "wins" (remember, upper bounds).

How about $\Omega\big(f(n) + g(n)\big)$? Here the function that grows slower "wins" (remember, lower bounds).

How about $\Theta\big(f(n) + g(n)\big)$? If $f(n) = \Theta\big(g(n)\big)$ then $\Theta\big(f(n) + g(n)\big) = \Theta\big(f(n)\big)$. Otherwise, the lower and upper bounds of $f(n) + g(n)$ do not grow in the same way, and so the Big Theta notation cannot be used.

How about a multiplication by a (positive) constant c? Easy. The constant is discarded, as it is implicit in the notation: $O\big(cf(n)\big) = O\big(f(n)\big)$, $\Omega\big(cf(n)\big) = \Omega\big(f(n)\big)$, and $\Theta\big(cf(n)\big) = \Theta\big(f(n)\big)$.

How about the product of two functions? Easy. Products are retained: $O\big(f(n)g(n)\big) = O\big(f(n)\big)O\big(g(n)\big)$, $\Omega\big(f(n)g(n)\big) = \Omega\big(f(n)\big)\Omega\big(g(n)\big)$, and $\Theta\big(f(n)g(n)\big) = \Theta\big(f(n)\big)\Theta\big(g(n)\big)$.

Asymptotic notation (part 3, useful information)

To determine the truth or falsewood of each of the statements $f(n) = O\big(g(n)\big)$, $f(n) = \Omega\big(g(n)\big)$, $f(n) = \Theta\big(g(n)\big)$, $f(n) = o\big(g(n)\big)$, one usually only needs to find out how $\frac{f(n)}{g(n)}$ behaves when n grows to infinity. In pathological cases where $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist, one will need to compute $\liminf_{n\to\infty} \frac{f(n)}{g(n)}$ and $\limsup_{n\to\infty} \frac{f(n)}{g(n)}$. We will not encounter these cases in this course. We have:

$$\lim_{n\to\infty}\left|\frac{f(n)}{g(n)}\right| = \begin{cases} 0, & \Rightarrow\ f(n) = o\big(g(n)\big) \quad f(n) = O\big(g(n)\big) \\ > 0, & \Rightarrow \qquad\qquad\quad f(n) = O\big(g(n)\big) \quad f(n) = \Theta\big(g(n)\big) \quad f(n) = \Omega\big(g(n)\big) \\ \infty, & \Rightarrow \qquad\qquad\quad\qquad\qquad\qquad\qquad\quad f(n) = \Omega\big(g(n)\big) \end{cases}$$

Informally, $f(n) = O\big(g(n)\big)$ when $f(n)$ does not grow faster than $g(n)$.

While mentally comparing two functions, one can discard all constants and all lower order terms. Note that n^a grows faster than $\log n$ for any $a > 0$, that n^a grows faster that n^b when $a > b \geqslant 0$, that a^n grows faster that b^n when $a > b \geqslant 1$, that a^n grows faster that n^b for any $a > 1$, and that $n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \frac{1}{12n}\right)$ (this is part of Stirling's asymptotic formula for $n!$) grows faster than any n^a or a^n. Note, however, that $n!$ grows slower than n^n. For example, in the comparison of

$$f(n) = 100n^3 + 10^{1000}n^{14/5} \qquad \text{with} \qquad g(n) = 10^{-1000}n! + 2^n$$

it is enough to compare

$$n^3 \qquad \text{with} \qquad n!$$

This is so because n^3 grows faster that $n^{14/5} = n^{2.8}$, and because $n!$ grows faster than any power. In this case $n!$ wins by a huge margin, and so $f(n) = o\big(g(n)\big)$. Of course, this also means that $f(n) = O\big(g(n)\big)$. In this particular case, when $n \leqslant 810$, $f(n)$ is actually larger than $g(n)$.

Asymptotic notation (part 4, extra notation)

In the previous slide it is stated that $\log n$ grows slower than any power of n. Mathematically, we say that $\log n = o(n^\epsilon)$ for any $\epsilon > 0$, or that $\log n = n^{o(1)}$. Sometimes, logarithmic factors are a nuisance, as they are minor details in a computational complexity expression. For example, the difference between $O(n^6)$ and $O(n^6 \log n)$ is just the relatively small $\log n$. In those cases it is possible to also hide the logarithmic factors behind the asymptotic notation. The $\tilde{O}$ notation was created for that purpose.

- **[Big Soft Oh notation]**

  $$f(n) = \tilde{O}\big(g(n)\big)$$

  means that

  $$f(n) = O\big(g(n) \log^k g(n)\big)$$

  for some $k > 0$.

Asymptotic notation (part 5, examples)

The following examples may help understand better the properties of the Big Oh notation:

- $10n^2 + 30n + 13 = O(n^2)$, because for $n \geqslant 31$ we have $10n^2 + 30n + 13 \leqslant 11n^2$. We have chosen $C = 11$, thus forcing n_0 to be at least 31. Any value of C larger than 10 would also work, but the closer it gets to 10 the larger n_0 has to be.
- $10n^2 + 30n + 13 = O(n^3)$, because for $n \geqslant 13$ we have $10n^2 + 30n + 13 \leqslant n^3$. We have chosen $C = 1$. In this case any positive value of C would also work.
- $10n^2+30n+13 \neq O(n)$, because no matter how C is chosen there exists an n for which $10n^2+30n+13 > Cn$.

We also have [**Homework:** explain why]:

- $10n^2 + 30n + 13 = \Omega(n^2)$.
- $10n^2 + 30n + 13 \neq \Omega(n^3)$.
- $10n^2 + 30n + 13 = \Omega(n)$.

and

- $10n^2 + 30n + 13 = \Theta(n^2)$.
- $10n^2 + 30n + 13 \neq \Theta(n^3)$.
- $10n^2 + 30n + 13 \neq \Theta(n)$.

Asymptotic notation (part 6, dominance relations)

The following functions are ordered according to their growth rate: 1, $\log n$, $\sqrt{n}$, n, $n \log n$, n^2, n^3, 2^n, $n!$.

Consider a processor that can do 10^{10} arithmetic operations per second. (That lies within the capabilities of top end contemporary processors.) The following table presents the time it takes that processor to solve a problem requiring $\log n$, n, $n \log n$, n^2, n^3, 2^n, $n!$, and n^n arithmetic operations (s means seconds, h means hours, d means days, and y means years):

n	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3	2^n	$n!$	n^n
10	0.2ns	0.3ns	1.0ns	2.3ns	10ns	100ns	102ns	363μs	1s
20	0.3ns	0.4ns	2.0ns	6.0ns	40ns	800ns	105μs	7.7y	3.3×10^8y
30	0.3ns	0.5ns	3.0ns	10ns	90ns	2.7μs	107ms	8.4×10^{14}y	
40	0.4ns	0.6ns	4.0ns	15ns	160ns	6.4μs	110s		
50	0.4ns	0.7ns	5.0ns	20ns	250ns	12μs	1.3d		
60	0.4ns	0.8ns	6.0ns	25ns	360ns	22μs	3.7y		
10^2	0.5ns	1ns	10ns	46ns	1.0μs	100μs	4.0×10^{12}y		
10^3	0.7ns	3.2ns	100ns	691ns	100μs	100ms			
10^4	0.9ns	10ns	1.0μs	9.2μs	10ms	100s			
10^5	1.2ns	32ns	10μs	115μs	1.0s	1.2d			
10^6	1.4ns	100ns	100μs	1.4ms	100s	3.2y			
10^7	1.6ns	316ns	1.0ms	16ms	2.8h	3.2×10^3y			
10^8	1.8ns	1μs	10ms	184ms	11.6d				
10^9	2.1ns	3.2μs	100ms	2.1s	3.2y				

Asymptotic notation (part 7, wisdom of a master)

> I also must confess a bit of bias against algorithms that are efficient only in an asymptotic sense, algorithms whose superior performance doesn't begin to "kick in" until the size of the problem exceeds the size of the universe. ... I can understand why the contemplation of ultimate limits has intellectual appeal and carries an academic cachet; but in The Art of Computer Programming I tend to give short shrift to any methods that I would never consider using myself in an actual program.
>
> — Donald E. Knuth, The Art of Computer Programming, preface of volume 4A (2011)

Classes of problems

In computer science problems are subdivided into classes:

- Problems that can be solved in polynomial time, i.e., for which there exits an $O(n^k)$ algorithm, are considered the tractable problems. They are said to belong to the class P (the P stands for **p**olynomial-time).
- There exist also some problems whose solution can be verified in polynomial time. Consider for example the problem of factoring an integer: finding its factors is believed to be a difficult problem, but verifying the factorization can be done using multiplications and primality tests, operations that are known to be in P. (Primality testing was only proved to be in P in 2002!) Problems of this type are said to belong to the class NP (NP stands for **N**ondeterministic **P**olynomial-time). It is not known if P=NP.
- There exists an important subset of NP problems that are equivalent to a certain "prototype" problem (the so-called boolean satisfiability problem, usually abbreviated as the SAT problem). Finding an efficient solution to one such problem automatically provides an efficient solution to the rest of them. These are said to belong to the class NP-complete. Many meaningful problems are known to belong to this class.

Useful formulas

The following formulas are useful to analyze algorithms:

- $\sum_{k=1}^{n} 1 = n$
- $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2$
- $\sum_{k=1}^{n} \frac{1}{k} = \log n + \underbrace{\gamma}_{\substack{\text{Euler's constant} \\ \approx 0.577216}} + \frac{1}{2n} + O(n^{-2})$
- $\sum_{k=n}^{m} f(k) = \sum_{k=1}^{m} f(k) - \sum_{k=1}^{n-1} f(k)$

A summation of the form $\sum_{k=a}^{b-1} f(k)$, with a and b integers, can be approximated by an integral of the form $\int_a^b f(x)\,dx$. More precisely, we have (Euler-Maclaurin summation formula):

$$\sum_{k=a}^{b-1} f(k) = \int_a^b f(x)\,dx + \sum_{k=1}^{m} \frac{B_k}{k!} f^{(k-1)}(x)\Big|_a^b + R_m,$$

where B_k are the Bernoulli numbers ($B_0 = 1$, $B_1 = -\frac{1}{2}$, $B_2 = \frac{1}{6}$, $B_3 = 0$, $B_4 = -\frac{1}{30}$, ...), are where

$$R_m = (-1)^{m+1} \int_a^b \frac{B_m(x - \lfloor x \rfloor)}{m!} f^{(m)}(x)\,dx$$

(here $\lfloor x \rfloor$ is the greatest integer less than or equal to x, so that $x - \lfloor x \rfloor$ is the fractional part of x, and $B_m(x)$ is the m-th order Bernoulli polynomial. [**Exercise**: confirm the first four formulas given above.]

Least squares fit (part 1, theory)

Suppose you have experimental data about the execution time of a program, in the form of $(x, y(x))$ pairs, where x is the problem size and $y(x)$ is the corresponding execution time. Suppose further that the theory tells you that the execution time should be of the form $Ax^2 + Bx + C$. How can you estimate A, B, and C?

To be more precise, suppose you have n data pairs (x_i, y_i), for $1 \leqslant i \leqslant n$. Suppose further that you know that you should have $y(x) = Ax^2 + Bx + C$, and that your data may have (hopefully small) errors e_i in the y_i values, so that $y_i = y(x_i) + e_i$. We may then write the system of equations

$$\underbrace{\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \\ \vdots & \vdots & \vdots \\ x_n^2 & x_n & 1 \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} A \\ B \\ C \end{bmatrix}}_{\mathbf{w}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ t_n \end{bmatrix}}_{\mathbf{y}} - \underbrace{\begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ \vdots \\ e_n \end{bmatrix}}_{\mathbf{e}},$$

which, using matrices and vectors, can be written in the compact form $\mathbf{Xw} = \mathbf{y} - \mathbf{e}$. One common technique to solve problems of this kind consists of minimizing the sum of the squares of the e_i errors. This is the essence of the so-called least squares method. The most robust way to solve problems of this type uses the singular value decomposition[1] of the $\mathbf{X}$ matrix, or, more precisely, the pseudo-inverse $\mathbf{X}^+$ of the $\mathbf{X}$ matrix[2]. The vector $\mathbf{w}_{\mathbf{opt}}$ that minimizes the sum of the squares of the errors, and that has the smallest length (if by chance there exist more that one solution) is given by

$$\mathbf{w}_{\mathbf{opt}} = \mathbf{X}^+\mathbf{y}.$$

[1] The singular value decomposition (SVD) is a generalization to rectangular matrices of the eigenvalue decomposition (that can only be used for square matrices).
[2] Like the SVD, the pseudo-inverse extends the concept of the inverse of a matrix to rectangular matrices; it is denoted by a superscript of $^+$.

Least squares fit (part 1, practice)

The best least squares solution can be found, using the `matlab` program, or its clone `octave`, in the following way:

- First, prepare a text file with your data. Put the x_i in the first column and the y_i in the second column. In the example below, we will use a file named `data.txt` with the contents

```
1  4
2  9
3 18
4 31
```

- Finally, we enter the following commands in `matlab` (assuming that $y(x) = Ax^2 + Bx + C$):

```
load data.txt
x = data(:,1);              % extract the first column
y = data(:,2);              % extract the second column
X = [ x.^2, x, 0*x+1 ];     % build the X matrix
w = pinv(X)*y;              % optimal solution (could also be written as w = X \ y;)
e = y-X*w;                  % optional: compute the errors vector
format long
w                           % print w --- A = w(1), B = w(2), and C = w(3)
norm(e)                     % optional: print the norm of the error vector (square root of the sum of squares)
plot(x,y,'.r',x,X*w,'og'); % plot the original data and its best least squares approximation
```

- For the data file given above you should get `w = [ 2,-1,3 ]` and a norm of the error vector very close to zero. (Because in this case $y(x) = 2x^2 - x + 3$.)

Of course, you should adapt the code given above to fit your $y(x)$ function. For example, if $y(x) = Ax^2 + Bx\log(x) + Cx + D$, then you should use

```
X = [ x.^2, x.*log(x), x, 0*x+1 ];
```

A first example

We will now determine the computational complexity of the following simple algorithm:

Algorithm: Linear search of unordered data.

Input: a sequence of n distinct numbers $a_1, a_2, \ldots, a_n$, with $n > 0$, and a number b.

Output: the smallest index i such that $a_i = b$, or 0 if no such index can be found.

Steps:

1. [Initialize index.] Set k to 1.
2. [Test.] If $a_k = b$ terminate the algorithm, with k as the output.
3. [Advance.] If $k < n$ then increase k and return to step 2. Otherwise terminate the algorithm with 0 as the output.

Let $f(n)$ denote the number of steps taken by the algorithm, and let $g(n)$ denote the number of average steps. The best case occurs when $b = a_1$. So, $f(n) = \Omega(1)$. The worst case occurs when b is different for all the a_i. In that case steps 2 and 3 will be done n times. Hence, $f(n) = O(n)$.

The average case depends on the probabilities p_i of the events $b = a_i$. Let $p_0 = 1 - \sum_{i=1}^{n} p_i$ be the probability of the remaining event, that b is different for all the a_i. When the algorithm terminates in step 2 for a certain value of i the total number of steps it performs is $1 + 2i + 3(i-1)$. That event has probability p_i. When the algorithm terminates in step 3, with probability p_0, the total number of steps it performs is $1 + 2n + 3(n-1) + 2$. We then have $g(n) = (5n)p_0 + \sum_{i=1}^{n}(5i-2)p_i$. When $p_i = \frac{1}{n}$ we have $p_0 = 0$ and $g(n) = \frac{1}{n}\sum_{i=1}^{n}(5i-2)$. This can be simplified (see next slide) to $g(n) = \frac{5n+1}{2}$, and so in this case $g(n) = O(n)$. In the general case $g(n)$ is as small as possible when $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$.

We could have counted only the number of times the body of the loop (steps 2 and 3) is performed. The result, $np_0 + \sum_{i=1}^{n} ip_i$, is similar to what we get from the more detailed analysis (but without the factor of 5).

More examples (part 1, $O(n)$ algorithms)

Examples of $O(n)$ algorithms:

- sum of the elements of a vector

```
double vector_sum(unsigned int n,
                  double a[n])
{
  double r = a[0];
  for(unsigned int i = 1u;i < n;i++)
    r += a[i];
  return r;
}
```

- inner product of two vectors

```
double vector_inner_product(unsigned int n,
                            double a[n],
                            double b[n])
{
  double r = a[0] * b[0];
  for(unsigned int i = 1u;i < n;i++)
    r += a[i] * b[i];
  return r;
}
```

- sum of two vectors:

```
void vector_addition(unsigned int n,
                     double a[n],
                     double b[n],
                     double r[n])
{ // r = a + b
  for(unsigned int i = 0u;i < n;i++)
    r[i] = a[i] + b[i];
}
```

- find the first index `i` for which `a[i]` is equal to `x`

```
unsigned int find_index(unsigned int n,
                        unsigned int a[n],
                        unsigned int x)
{ // returns n when x is not found
  unsigned int i;

  for(i = 0u;i < n && a[i] != x;i++)
    ;
  return i;
}
```

- count out many times `a[i]` is equal to `x`

```
unsigned int count_indices(unsigned int n,
                           unsigned int a[n],
                           unsigned int x)
{
  unsigned int c = 0u;

  for(unsigned int i = 0u;i < n;i++)
    if(a[i] == x)
      c++;
  return c;
}
```

- recursive computation of $n!$

```
double factorial(unsigned int n)
{
  return (n < 2u) ? 1.0 : (double)n * factorial(n - 1u);
}
```

More examples (part 2, $O(n^2)$ algorithms)

Examples of $O(n^2)$ algorithms:

- multiplication of multi-precision integers (one base-10 digit per array entry)

```
void carry_propagation(unsigned int n,
                       unsigned int a[n])
{
  unsigned int carry = 0u;
  for(unsigned int i = 0u;i < n;i++)
  {
    carry += a[i];
    a[i] = carry % 10u;
    carry /= 10u;
  }
  assert(carry == 0u);
}
void multiplication(unsigned int n,
                    unsigned int a[n],
                    unsigned int b[n],
                    unsigned int r[2u * n])
{ // r = a * b
  assert(sizeof(n) >= (size_t)4 && n <= 47721858u);
  for(unsigned int i = 0u;i < 2u * n;i++)
    r[i] = 0u;
  for(unsigned int i = 0u;i < n;i++)
    if(a[i] != 0u)
      for(unsigned int j = 0u;j < n;j++)
        r[i + j] += a[i] * b[j];
  carry_propagation(2u * n,&r[0]);
}
```

[**Homework:** Why 47721858?]

- sum of two matrices

```
void matrix_addition(unsigned int n,
                     double A[n][n],
                     double B[n][n],
                     double R[n][n])
{ // R = A + B
  unsigned int i,j;

  for(i = 0u;i < n;i++)
    for(j = 0u;j < n;j++)
      R[i][j] = A[i][j] + B[i][j];
}
```

- insertion sort

```
void insertion_sort(unsigned int n,
                    double a[n])
{
  unsigned int i,j;
  double v;

  for(i = 1u;i < n;i++)
  {
    v = a[i];
    for(j = i;j > 0u && v < a[j - 1u];j--)
      a[j] = a[j - 1u];
    a[j] = v;
  }
}
```

[**Homework:** What are the best and worst cases?]

More examples (part 3, improved multiplication)

The multiplication of two integers, A and B, each with with $2n$ bits, can be done as follows.

- Split A into two halves, A_1 and A_0, each with n bits, so that $A = A_1 2^n + A_0$.
- Likewise, split B into two halves, B_1 and B_0, again each with n bits, so that $B = B_1 2^n + B_0$.
- In the standard multiplication method the product of A and B is computed with the formula $(A_1 B_1)2^{2n} + (A_1 B_0 + A_0 B_1)2^n + (A_0 B_0)$. This requires 4 multiplications of numbers with half the number of bits, so, according to the master theorem, presented later in this course, using this formula recursively gives rise to an $O(n^2)$ algorithm to compute the product.
- Doing more additions and subtractions, which are $O(n)$ operations, it is possible to eliminate one multiplication by taking advantage of the identity $A_1 B_0 + A_0 B_1 = (A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0$. Its right hand side can be computed using only one extra multiplication (the products $A_1 B_1$ and $A_0 B_0$ can be reused).
- Thus, if $T(n)$ is the time required to multiply two n-bits integers, we have $T(2n) \leqslant 3T(n) + \alpha n$, where the αn term captures the time required to do additions, subtractions, and house-keeping tasks. It follows from the master theorem that $T(n) = O(n^{\log_2 3})$. Note that $\log_2 3 \approx 1.58496$ is considerably smaller than 2, so this simple method is a substantial improvement over the original method. [**Homework:** Compare n^2 with $n^{\log_2 3}$ for $n = 10^k$, $k = 1, 2, 3, 4, 5, 6$.]
- Using more advanced methods it is possible to multiply two integers much faster than this. The best known method does the job in $O(n \log n)$ steps and uses FFTs (Fast Fourier Transforms).

More examples (part 4, $O(n^3)$ algorithms)

Examples of $O(n^3)$ algorithms:

- multiplication of two matrices

```
void matrix_matrix_product(unsigned int n,
                           double A[n][n],
                           double B[n][n],
                           double R[n][n])
{ // R = A * B
  for(unsigned int i = 0u;i < n;i++)
    for(unsigned int j = 0u;j < n;j++)
      R[i][j] = 0.0;
  for(unsigned int i = 0u;i < n;i++)
    for(unsigned int j = 0u;j < n;j++)
      for(unsigned int k = 0u;k < n;k++)
        R[i][j] += A[i][k] * B[k][j];
}
```

  [**Homework:** The three nested `for` loops can be done in any order. Measure the execution times, for $n = 511$, $n = 512$, and $n = 513$, for the 6 possible orders of the loops (ijk, ikj, jik, jki, kij, and kji.]

- matrix inversion (not much different from the next algorithm).

- determinant of a matrix

```
double matrix_determinant(unsigned int n,
                          double A[n][n])
{ // warning: A is modified
  unsigned int i,j,k;
  double r,t;

  r = 1.0;
  for(i = 0u;i < n;i++)
  {
    // find the biggest element (the pivot)
    j = i;
    for(k = i + 1u;k < n;k++)
      if(fabs(A[k][i]) > fabs(A[j][i]))
        j = k;
    // exchange lines (if necessary)
    if(j != i)
      for(r = -r,k = i;k < n;k++)
      {
        t = A[i][k];
        A[i][k] = A[j][k];
        A[j][k] = t;
      }
    // Gauss-Jordan elimination
    for(r *= A[i][i],j = i + 1u;j < n;j++)
      for(k = i + 1u;k < n;k++)
        A[j][k] -= (A[j][i] / A[i][i]) * A[i][k];
  }
  return r;
}
```

More examples (part 5, improved matrix multiplication)

The matrix multiplication of a $(2n) \times (2n)$ matrix can be split as follows:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

This yields 8 multiplications and 4 additions of $n \times n$ matrices. Doing this recursively gives rise to a $O(n^3)$ algorithm. Since the computational complexity of a matrix addition is way smaller than that of a matrix multiplication, the number of additions is irrelevant in theory (but not in practice).

Strassen found a way to compute

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

that only requires 7 multiplications (and 18 additions). Winograd later reduced the number of additions to 15. The product is given by

$$\begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & W + V + (A_{11} + A_{12} - A_{21} - A_{22})B_{22} \\ W + U + A_{22}(B_{21} + B_{12} - B_{11} - B_{22}) & W + U + V \end{bmatrix}$$

where $U = (A_{21} - A_{11})(B_{12} - B_{22})$, $V = (A_{21} + A_{22})(B_{12} - B_{11})$, and $W = A_{11}B_{11} + (A_{21} + A_{22} - A_{11})(B_{11} + B_{22} - B_{12})$. This gives rise to a $O(n^{\log_2 7}) \approx O(n^{2.807})$ algorithm to multiply matrices. With considerable effort, it is possible to reduce the exponent to a number closer to 2. The current record is an exponent of only 2.37286. Note, however, that all known methods with an exponent smaller that 2.7 are so complex, and with horrendous proportionality constants, that they are **useless** in practice (reread the wisdom of a master)!

More examples (part 6a, exponential complexity)

The following algorithms have exponential complexity:

- computing Fibonacci numbers (in a dumb way)

```
double F(unsigned int n)
{
  return (n < 2u) ? (double)n : F(n - 1u) + F(n - 2u);
}
```

Homework: what is the computational complexity of this function? How about the computational complexity of this "improved" way of computing Fibonacci numbers?

```
double Fi(unsigned int n)
{
  static double Ft[10] =
  {
     0.0, 1.0, 1.0, 2.0, 3.0, // 0 to 4
     5.0, 8.0,13.0,21.0,34.0  // 5 to 9
  };

  return (n < 10u) ? Ft[n] : Fi(n - 1u) + Fi(n - 2u);
}
```

- print all possible sums of the elements of an array (generate all subsets); it is assumed that `n > 0` and that `n < 8*sizeof(unsigned long)`.

```
void print_all_sums(unsigned int n,double a[n])
{
  unsigned long mask;
  unsigned int i,j;
  double sum;

  mask = 0ul; // if the bit number i is set to 0 then
              // a[i] will not contribute to the sum
              // we begin with the empty set
  do
  {
    // do sum
    sum = 0.0;
    for(i = j = 0u;i < n;i++)
      if(((mask >> i) & 1ul) != 0ul)
      {
        sum += a[i];
        printf("%sa[%u]",(j == 0u) ? "" : "+",i);
        j = 1u; // next time print a + sign
      }
    printf("%s = %.3f\n",(j == 0u) ? "empty" : "",sum);
    // next subset (discrete binary counter)
    mask++;
  }
  while(mask < (1ul << n));
}
```

More examples (part 6b, print all sums done in a recursive way)

The `print_all_sums` function code of the previous slide uses a well known trick to represent subsets of a set with a relatively small number of elements. The same function can also be implemented in a recursive way, as illustrated by the following code.

```
void print_all_sums_recursive(unsigned int n,unsigned int m,double a[n],double sum,unsigned long mask)
{
  if(m == n)
  { // nothing more to do; print sum
    unsigned int i,j;

    for(i = j = 0u;i < n;i++)
      if(((mask >> i) & 1ul) != 0ul)
      {
        printf("%sa[%u]",(j == 0u) ? "" : "+",i);
        j = 1u; // next time print a + sign
      }
    printf("%s = %.3f\n",(j == 0u) ? "empty" : "",sum);
  }
  else
  {
    print_all_sums_recursive(n,m + 1u,&a[0],sum       ,mask             ); // do not use a[m]
    print_all_sums_recursive(n,m + 1u,&a[0],sum + a[m],mask | (1ul << m)); // use a[m]
  }
}
```

At the top level, this function should be called with `m = 0`, `sum = 0.0`, and `mask = 0ul`. At each level of the recursion, the code tries two possibilities: either the sum includes `a[m]` or it does not. The recursion terminates when `m = n`.

If it is not necessary to keep track of the terms that are part of the sum then the `mask` argument is not needed.

More examples (part 7a, worst than exponential complexity)

When called with `m` equal to `0`, the following recursive function prints all permutations of the integers a_0, a_1, ..., a_{n-1}, assumed to be distinct and to be stored in the array `a[]`:

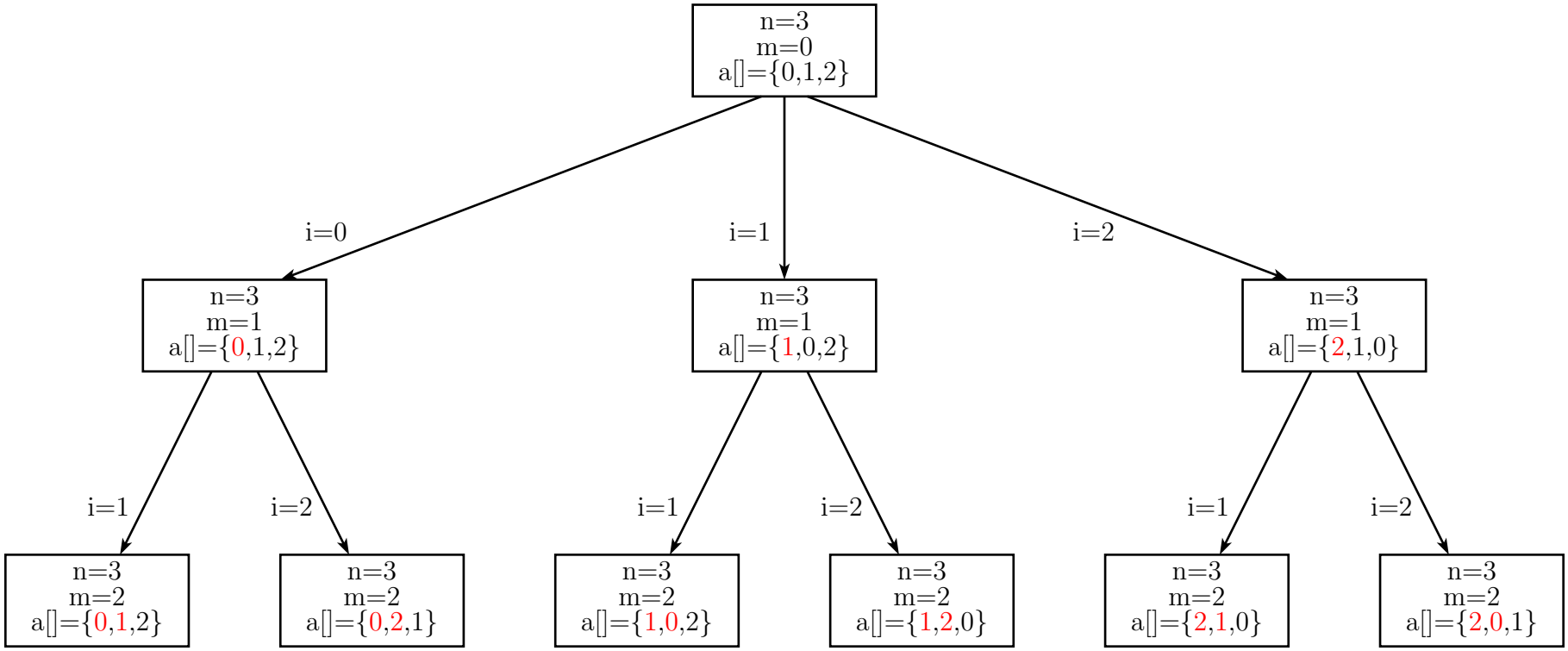
```
void print_all_permutations_recursive(unsigned int n,unsigned int m,int a[n])
{
  unsigned int i;

  if(m == n - 1u)
  { // nothing more to do, visit permutation
    for(i = 0u;i < n;i++)
      printf("%s%d",(i == 0u) ? "" : " ",a[i]);
    printf("\n");
  }
  else // not yet at the end, place in a[m] each remaining integer and recurse
    for(i = m;i < n;i++)
    {
#define swap(i,j)  do { int t = a[i]; a[i] = a[j]; a[j] = t; } while(0)
      swap(i,m);                                          // swap a[i] with a[m]
      print_all_permutations_recursive(n,m + 1u,&a[0]); // recurse
      swap(i,m);                                          // undo the swap of a[i] with a[m]
#undef swap
    }
}
```

When called with `m` larger than `0` this function does not change the values of `a[0]`, `a[1]`, ..., `a[m-1]`, and places in `a[m]` each of the remaining values before calling itself recursively. When `m` is equal to `n-1` there is nothing left to do so the permutation is visited.

Note that although inside the function the contents of the `a[]` array are reordered, when it returns they fall back to their original values. The computational complexity of this function is $O(n \times n!)$, because the block that prints each permutation is visited $n!$ times, and printing the permutation is $O(n)$.

More examples (part 7b, `print_all_permutations_recursive` example)

When called with `n=3`, `m=0`, and `a[]={0,1,2}`, the call tree of the `print_all_permutations` function is the following:



As we go deeper in the call tree, the red part of the `a[]` array does not change. As can be seen in this small example, the permutations are **not** generated in lexicographic order.

It is possible to prune parts of this call tree. To discard a sub-tree, just to not call the function recursively at the appropriate place.

More examples (part 8, algorithms with "small" computational complexity)

We conclude this tour of examples with two algorithms that have small computational complexity:

- computation of x^y using a mathematical formula (y is a real number)

```
double power_dd(double x,double y)
{
  return exp(y * log(x));
}
```

  This is an $O(1)$ algorithm. It works only when $x > 0$.

- computation of x^n using recursion (n is an integer)

```
double power_di(double x,int n)
{
  if(n < 0)  return power_di(1.0 / x,-n);
  if(n == 0) return 1.0;
  double t = power_di(x * x,n / 2); // the integer division discards a fractional part
  return (n % 2 == 0) ? t : x * t;  // take care of the discarded fractional part
}
```

  This is an $O(\log n)$ algorithm. With the exception of $x = 0$ and $n < 0$, it works for any x. (By convention $0^0 = 1$; 0 raised to a negative power is illegal.)

- computation of $a^b \bmod c$ (u32 is `unsigned int` and u64 is `unsigned long long`)

```
u32 power_mod(u32 a,u32 b,u32 c)
{
# define mult_mod(x,y,m)  (u32)(((u64)(x) * (u64)(y)) % (u64)(m))  // (x*y) mod m
  if(b == 0) return 1u;
  u32 t = power_mod(mult_mod(a,a,c),b / 2,c);
  return (b % 2 == 0) ? t : mult_mod(a,t,c);
# undef mult_mod
}
```

Computational challenge

The $\mathbf{fusc}$ function – so named by Edsger W. Dijkstra — is defined in the following way for all non-negative integers:

- for $n = 0$ or $n = 1$, $\mathbf{fusc}(n) = n$, and
- for $n \geqslant 1$, $\mathbf{fusc}(2n) = \mathbf{fusc}(n)$ and $\mathbf{fusc}(2n+1) = \mathbf{fusc}(n) + \mathbf{fusc}(n+1)$.

This function is related to the so-called Calkin-Wilf tree, which is a clever way of generating all rational numbers. It is known that $\mathbf{fusc}(n) = O\big(n^{\log_2((1+\sqrt{5})/2)}\big) \approx O(n^{0.69424})$.

Task 1: write a computer program capable of computing $\mathbf{fusc}(n)$ for any 64-bit unsigned integer.

Task 2: how long does it take to compute $\mathbf{fusc}(750599848416597)$?

Task 3: and how long does it take to compute $\mathbf{fusc}(12297829381041378645)$? [**Hint:** are you a masochist?]

Task 4: for your program, is it true that the time it takes to compute $\mathbf{fusc}(n)$ is $O\big(\mathbf{fusc}(n)\big)$?

Task 5: find all record-breaking values of $\mathbf{fusc}(n)$ for $n \leqslant 10^9$; a record-breaking value of a function $f(n)$ is a value of $f(n)$ for which $f(m) < f(n)$ for all $m < n$. Can you find any regularity in the values of n of the record-breakers? Taking advantage of what you have found, give a high-probability list of all record-breakers for $n < 2^{64}$. [**Hint:** this is hard, do task 6 first. If you did not find any regularity, you are done for.]

Task 6: $\mathbf{fusc}(n)$ can be computed much more efficiently by writing recursion formulas for the expression

$$a \times \mathbf{fusc}(n) + b \times \mathbf{fusc}(n+1).$$

You need to consider two cases: n even and n odd. Rewrite your computer program to take advantage of these formulas. What is the computational complexity of the new program? How much time does it take to compute $\mathbf{fusc}(12297829381041378645)$ with the new program?

What is the value of $\mathbf{fusc}(226854911280625642302767490263275623765)$?