1 Prologue

1.1 Algorithm

Johann Gutenberg’s discovery allowing him to print books by putting together metallic pieces revolutionized the world. It made books widely accessible. It affected literacy, education, political thought, humanities and science.

But there are others who think that not typography but algorithms was the key development. How to deal with large numbers (of critical importance to accounting, business, trade, science, and engineering) was not a trivial matter. Indeed, consider the question how to add or multiply Roman numerals (say add MCDXII and DCCCXII)? And there are more complex problems, too, like finding square roots, approximating irrational numbers, etc. The discovery of positional systems was a major breakthrough. The sexagesimal system was invented by Summerians around 2000 BCE. The decimal system was known (or was in some use) very early on (perhaps as early as 3000 BCE) but in its modern form, with 0 (zero) was invented (popularized) around 600 CE in India (c. 598-670, Brahmagupta explains the Hindu-Arabic numerals).

Important to us is the fact that Muhammad ibn Musa al-Kwarizmi (the first half of the 9th century) for the first time described and popularized the decimal system as well as basic procedures for manipulating decimal numbers outside of India. These procedures were mechanical, precise, efficient and correct. Ultimately, a few centuries later, Europe adopted the system and the rest is history. Mathematics and science took off. The terms algorism and more modern algorithm reflect the name of al-Kwarizmi — the Arabic bridge from India to Europe.

1.2 Fibonacci

Leonardo Fibonacci of Pisa was the 13th century Italian mathematician who was largely responsible for popularizing the decimal system in the west. He also made other contributions, for which he is possibly even more broadly known today. Most notable is the
sequence
0, 1, 1, 2, 3, 5, 8, 13, …

known as the Fibonacci sequence. No other sequence has such broad applications (biology, demography, art, architecture, music, computer science). For instance the numbers of intersecting spirals on cones are consecutive Fibonacci numbers and most spiral shells follow the golden spiral pattern (just google it!). Importance of the sequence stems from the fact that it is closely related to the golden ratio.

![Diagram of Fibonacci sequence]

In CS, besides the sequence of the powers of 2, the Fibonacci sequence is CS’s favorite sequence.

We will use it to introduce and illustrate some key ideas related to algorithms. The Fibonacci sequence can be defined by the simple recursive rule:

\[
F_n = \begin{cases} 
F_{n-1} + F_{n-2} & \text{if } n \geq 2 \\
1 & \text{if } n = 1 \\
0 & \text{if } n = 0.
\end{cases}
\]

The numbers grow quickly. In fact, \(F_n = \lfloor (\frac{1 + \sqrt{5}}{2})^n \rfloor \approx 2^{0.694n}\). But what is the exact value of \(F_{100}\), or of \(F_{200}\)? To answer, we need an algorithm for computing the \(n\)th Fibonacci number.

1.2.1 The first attempt

```python
function fib1(n)
    if n = 0: return 0
    if n = 1: return 1
    return fib1(n-1) + fib1(n-2)
```

This is indeed a mechanical, unambiguous procedure (thus, roughly, an algorithm). Four key questions arise (that pertain to any algorithm).

1. Does it do what it is meant to do? Or, equivalently, is it correct?

2. How much time does it take?
3. How much space does it take? This one will be of somewhat lesser importance to us in this course.

4. Can we do better?

Clearly, the algorithm described here is correct! It is just a rewriting of the definition of the \(n\)th Fibonacci number in \textit{pseudo-code}.

To answer the second question, let \(T(n)\), \(n \geq 1\), be the number of \textit{conceptual steps} needed to compute \(\text{fib1}(n)\). Then

\[
T(0) = 2 \\
T(1) = 3
\]

For larger values of \(n\) (\(n \geq 2\)),

\[
T(n) = T(n - 1) + T(n - 2) + 6
\]

Clearly, for every \(n \geq 1\), \(T(n) \geq F_n\). This is very bad news indeed. The time is \textit{exponential} in \(n\), and double exponential(!) in the size of the decimal or, better yet, binary representation of \(n\).

Just how bad is that? \(T(200) \geq F_{200} \approx 2^{138}\). According to \url{http://www.top500.org/lists/2015/06/} (accessed on January 13, 2016)

For the fifth consecutive time, Tianhe-2 (Milkyway-2), a supercomputer developed by Chinas National University of Defense Technology, has retained its position as the worlds No. 1 system, according to the 45th edition of the twice-yearly TOP500 list of the worlds most powerful supercomputers. Tianhe-2, which means Milky Way-2, led the list with a performance of 33.86 petaflop/s (quadrillions of calculations per second) on the Linpack benchmark.

[MT comment: in 2008, the fastest supercomputer clocked at about one petaflop (\(10^{15}\) calculations per second). This is a huge improvement but still very far away from the exaflop - \(10^{18}\) calculations per second]

We have \(2^{138} > 10^{41}\) (as \(\log_2 10 \approx 3.32\)). Thus, the Tianhe-2 computer would require more than

\[
10^{41}/(33.86 \times 10^{15}) = 2.95 \times 10^{24}
\]

seconds to complete the task. BTW, the sun will turn into a red giant star (ending life on earth) in about 4.6 billion years (there is some uncertainty in that estimate). That is in

\[
4.6 \times 10^9 \times 365 \times 24 \times 3600 = 1.45 \times 10^{17}
\]

seconds! And getting a computer working at exaflop speeds will not help much!! No, we need another method if we want to know what \(F_{200}\) is! Can we do better?
1.2.2 A better method is possible!

The problem with the first method is that so many computations are repeated. Could we save them and reuse? Yes, we can!

```python
function fib2(n)
if n = 0: return 0
create an array f[0..n]
f[0] = 0
f[1] = 1
for i = 2..n:
    f[i] = f[i-1] + f[i-2]
return f[n]
```

Looks that this algorithm runs in time that is linear in \( n \). That is much better. We can easily compute \( F_{200} \) now. But beware! to store the answer you need about 138 bits; a lot of attention to the arithmetic on “unbounded length” integers needed.

For instance, you might consider setting up three vectors of integers (or just bits or boolean values). Two of these vectors would contain \( F_{i-1} \) and \( F_{i-2} \). They would then be added (in binary) to form \( F_i \) to be stored in the third vector. And so on ... . Initially, the first two vectors would represent 0 and the second 1. Since the addition takes time that is linear in the length of the numbers to be added (in the number of bits or digits in the representation of the numbers), this algorithm really takes time that is better approximated by the formula

\[
\frac{n^2}{2}
\]

Indeed, there are \( n-1 \) iterations of the loop, in each of them one addition. In the worst case, the numbers to be added have \( 0.694 \times n \) bits. This is good. Computing \( F_{200} \) is certainly feasible. However, the algorithm is exponential in \( \log_2 n \) — the size of input. In particular, trying to compute to Fibonacci number \( F_i \), where \( i \) requires, say, 100 bits to represent (the size of input is 100) is hopeless.

**Comment:** Note that the formula \( n \log n \) is not accurate. But it captures the essence of the behavior of the algorithm — how well it scales with \( n \). We will return to this issue shortly.

1.3 Positional notation or place-value notation

An integer is an abstract concept. There are many ways to represent it. The simplest (but not the best) one is to use a sequence of symbols (say strokes \( | \) ) of length equal to the number. For instance, \(|||\) represents 4 and \(|||||\) represents 9 in this system. But, how to represent 0 or negative integers is not entirely clear. Moreover, it is completely impractical for representing larger integers (even as small as, day 257 or 2016). However, it does support easy algorithms for arithmetic operations (how would you add or multiply?).

Positional notation solves the problem of the size of the representation and also allows for good arithmetic algorithms. A positional notation requires one parameter, its base, an
integer $b \geq 2$. It also requires $b$ symbols to represent digits of the system. The digits of the binary system ($b = 2$) are 0 and 1. The digits of the decimal system ($b = 10$) are \{0, 1, \ldots, 9\}, and the digits of the hexadecimal systems are \{0, 1, \ldots, 9, A, B, C, D, E, F\}. Integers are represented as signed sequences of digits (for instance, 11101, 0, -10101 in binary). Since the digit of a binary system are also the digits of the decimal system, 10011 could be interpreted as ten thousand eleven (in decimal) or nineteen (in binary). Thus, to avoid confusion, it is important that the base is understood.

Given a system in base $b$ (and with the digits 0, 1, \ldots, $b-1$), zero is represented by the sequence 0, and non-negative numbers are represented as sequences $\langle a_{n-1}, a_{n-2}, \ldots, a_1, a_0 \rangle$ of digits, where $a_{n-1} > 0$. The meaning of this representation, that is, the integer the sequence stands for is given by

$$a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + \ldots + a_1b^1 + a_0b^0.$$ 

(In some cases, for convenience, we allow “padding” sequences representing integers with some number of leading 0’s. It is a valid operation in a sense that it does not affect the sum above, and so the “padded” sequence can be thought of as representing the same integer.)

### 1.4 Some useful math facts

Let $a$ and $b$ be positive reals, $a \neq 1$. Logarithm base $a$ of $b$ is defined as that (unique) real $c$ that satisfies $a^c = b$. In symbols,

$$\log_a b = c \quad \text{precisely when} \quad a^c = b.$$ 

The logarithm function satisfies the following fundamental identity: For every $a, c \neq 1$, $a, b, c > 0$

$$\log_a b = \log_a c \times \log_c b$$

or, equivalently,

$$\log_c b = \frac{\log_a b}{\log_a c}.$$ 

This is useful when we want to change the base of the logarithm. For instance,

$$\log_2 n = \log_2 10 \times \log_{10} n \approx 3.32 \log_{10} n.$$ 

Second, the very definition of the logarithm implies that

$$a = b^{\log_b a}.$$ 

This identity offers a way to represent functions of the form $f(n)^{g(n)}$ (with $f(n)$ positive) as $a^{g'(n)}$, where $a > 0$ and $a \neq 1$ is a constant (meaning that for any such constant $a$, a function $g'(n)$ can be found so that $f(n)^{g(n)} = a^{g'(n)}$). For instance,

$$n^{\log_2 n} = (2^{\log_2 n})^{\log_2 n} = 2^{\log^2 n}.$$ 

This allows us to compare exponential functions with different bases (discussed later).
1.5 Big-\(O\) notation

There is a danger of sloppiness in the analysis of the running time (what is the size of input, how close are our approximations). But the other extreme, of being too precise, also is dangerous. The point is that too precise an analysis may just be impossible to carry out. **But meaningful estimates are possible!!**

The right level of abstraction and simplification is what is needed. Focusing on computer steps is a simplification. But such analysis would depend to a large degree on processor architecture and would not generalize from one computer to the next. We need a machine-independent way to assess the running time of algorithms. This measure is the number of basic computer operations as a function of the size of the input. But we will simplify this measure further. We will focus on dominating terms of this functions, those that are responsible for how fast the function grows.

**Definition 1.1 (Asymptotic “less than or equal”)** Let \(f(n)\) and \(g(n)\) be functions from non-negative integers to reals such that \(f(n), g(n)\) are eventually positive.\(^2\) We say \(f = O(g)\) (\(f\) is big-\(O\) of \(g\), which means that \(f\) grows no faster than \(g\)) if there is a constant \(c > 0\) such that eventually \(f \leq c \cdot g\).

We can also define \(f = \Omega(g)\) for “asymptotic \(\geq\)” and \(f = \Theta(g)\), for “asymptotic equality.”

Properties:

1. If \(f = O(h)\) and \(g = O(h)\) then \(f + g = O(h)\) and \(fg = O(h^2)\) (and the same for other symbols)

2. Some care is needed when analyzing summations, where the number of terms is also a function of \(n\). For instance, for every \(k\), the constant function \(f_k\) such that \(f_k(n) = k\) satisfies \(f_k = O(1)\). Let us define \(g\) as follows:

   \[
g(n) = \sum_{i=1}^{n} f_i(n).
   \]

   Clearly, \(g(n) = n(n+1)/2 = \Theta(n^2)\). Thus, \(g(n) \neq O(n)\). But it might be tempting to reason as follows:

   \[
g(n) = \sum_{i=1}^{n} f_i(n) = \sum_{i=1}^{n} O(1) = O\left(\sum_{i=1}^{n} 1\right) = O(n).
   \]

   The result contradicts our earlier finding. The simplification does not work and leads to an error. The error comes from the fact that the constants involved in the identities \(f_i = O(1)\) and hidden behind the big-\(O\) notation are growing with \(k\). However, if there is a single constant \(c\) demonstrating for every \(i\) that \(f_i = O(1)\), the reasoning above would work.

\(^2\)A property of integers holds *eventually* if there is an integer \(n_0\) such that the property holds for every \(n \geq n_0\); here, the real function \(f\) defined on non-negative integers is *eventually positive* if the property \(f(n) > 0\) holds eventually.
To estimate the running time of an algorithm, $T(n)$ (where $n$ represents the size of input), we try to find a simple function $f(n)$ (for instance, $n^k$, $k^n$, $\log^p n$, $n^k \log^p n$, etc.) such that $T(n) = O(f(n))$. Moreover, the more slowly growing the function the better.

Here are some intuitive rules to use when simplifying functions (they can be made mathematically precise):

1. Drop multiplicative constants ($14n^2 \rightarrow n^2$)
2. $n^a$ dominates $n^b$ if $a > b$
3. $a^n$ dominates $b^n$ if $a > b > 0$
4. $n^k$ dominates $\log^p n$ for every $k, p > 0$

Typical functions we like to use to estimate the rate of growth of algorithms are $n^k$ (for instance, $n^2$), $n^k \log^p n$ (for instance, $n \log n$), $a^n$ (for instance, $2^n$ or $2^{n^2}$).

Finally, we have to realize that care is needed in estimating the size of the input. In most cases, the size is given by the number of basic input components (vertices and edges in the graph, numbers to sort, etc.). In such cases, it is okay to disregard the size of numbers that form the input, and assume that each occupies a single memory location and that each arithmetic and logical operation on numbers takes unit time.

In other cases, for instance for algorithms with numbers, the input may be just one or two numbers and the previous approach makes no sense as the measure of the input size. In those cases, we must use the number of bits in the binary representation of the input elements as the measure of the size.

Finally, we note that in most cases all we can do is only estimate the worst-case running time and so $\Theta()$ is rarely used in the context of time analysis.

Some problems to work out.

1. Provide an estimate of the running time for the algorithm that searches through a linked list for a record with a given key. The algorithm returns the first such record and stops, or runs through the entire list, outputs “not found” and stops. Here, big-Oh estimate possible, big-$\Theta$ estimate not possible.

2. Provide a big-Theta estimate of the running time of the following algorithm (expressed as a function of $n$; assume that $n$ is a non-negative integer).

   ```
   a = n
   while $n \geq 1$:
       $n = n - 1$
       $a = a + i$
   return $a$
   ```

3. Provide a big-Oh estimate of the running time of the following algorithm (expressed as a function of $n$; assume that $n$ is a non-negative integer). Your estimate should be asymptotically best possible.
\[ a = n \]
\[ \text{if } n \text{ is odd:} \]
\[ \text{while } n \geq 1: \]
\[ n = n - 1 \]
\[ a = a + i \]
\[ \text{return } n \]

Can you provide a big-Theta estimate for this algorithm? Explain.

4. Is \( 3^n = O(n!) \)? Is \( n! = O(3^n) \)?

5. Is \( 3^n = 5^{n/2} \)? Is \( 5^{n/2} = 3^n \)?

Solution: We have \( 3 = 5^{\log_5 3} \). Therefore,
\[
3^n = (5^{\log_5 3})^n = 5^{n \log_5 3} = 5^{0.683n}
\]

Consequently,
\[
\frac{3^n}{5^{n/2}} = 5^{0.183n} \rightarrow \infty.
\]

Thus, there is no constant \( c \) such that eventually \( 3^n \leq c5^{n/2} \) and so, \( 3^n \neq O(5^{n/2}) \).
However \( 5^{n/2} \leq 3^n \), Thus, if we take \( c = 1 \), we have that eventually (in fact, here, right from the start) \( 5^{n/2} \leq c3^n \). It follows that \( 5^{n/2} = O(3^n) \).

6. Prove that \( f + g = \Theta(\max\{f, g\}) \)

7. Compare \( n + (-1)^n n \) with \( n + (-1)^{n+1} n \)

8. Estimate the running time of an algorithm that given \( n \) numbers, returns the sum of all of them, if \( n \) is odd, and the sum of the first \( n/2 \) of them, if \( n \) is even.

9. Estimate the running time of insertion sort.

10. Find a simple function \( f \) capturing the rate of growth of \( 3n^3 \log n + 1.01^n + n^{\sqrt{n}} \).

That is, we must have \( 3n^3 \log n + 1.01^n + n^{\sqrt{n}} = \Theta(f) \)

Solution: \( 1.01^n \) being an exponential function with base greater than 1 asymptotically dominates any polynomial. In particular, it dominates \( 3n^4 \). Since \( n \) dominates \( \log n \), \( 3n^4 \) dominates \( 3n^3 \log n \). Thus \( 1.01^n \) dominates \( 3n^3 \log n \) and, consequently, the latter can be dropped (it does not affect the asymptotic growth of the function).

Further, \( n = 1.01^{\log_{1.01} n} \). Thus,
\[
n^{\sqrt{n}} = (1.01^{\log_{1.01} n})^{\sqrt{n}} = 1.01^{\sqrt{n} \log_{1.01} n}.
\]

Since \( n > 2 \log_{1.01} n \),
\[
\frac{1.01^n}{1.01^{\sqrt{n} \log_{1.01} n}} > 1.01^{\sqrt{n} \log_{1.01} n} \rightarrow \infty.
\]
Thus, $1.01^n$ dominates $1.01^{\sqrt{\pi \log_{1.01} n}}$. It follows that the term $1.01^{\sqrt{\pi \log_{1.01} n}}$ can be dropped, too. Thus,

$$3n^3 \log n + 1.01^n + n^{\sqrt{n}} = \Theta(1.01^n).$$

11. Find a simple function $f$ capturing the rate of growth of $3n^3 \log n + n^2 \log^4 n + n + 3 \log_2 n$. That is, we must have $3n^3 \log n + n^2 \log^4 n + n + 3 \log_2 n = \Theta(f)$.

## 2 Algorithms with numbers

We represent numbers as vectors (sequences) of their digits in the base that we choose. If the base is 2, the digits are 0 and 1. The number 6 is represented as $[0, 1, 1]$, where digit in the position 0 is the least significant digit. Following the same convention, 141 in base 4 would be represented as $[1, 3, 0, 2]$ because $141 = 1 \times 4^0 + 3 \times 4^1 + 0 \times 4^2 + 2 \times 4^3$. And, for one more example, 3709 in base 10 would be represented as $[9, 0, 7, 3]$. The key here is that digits are limited to the range $\{0, 1, \ldots, b-1\}$ (where $b$ stands for the base), but the vectors can be arbitrarily long and arbitrarily large numbers can be represented this way (without worry for overflows in typed languages, where there is a limit on the size of the maximum value for a variable of type integer).

We will assume that 0 is represented by the empty vector $[]$. We will also assume that when we need negative numbers, we will represent a non-zero integer by its absolute value and its sign.

We start by providing a simple algorithm to convert from binary to decimal. The input number is given as a vector of binary digits (the digit in position 0 being least significant). The output is a vector of decimal digits representing the same integer (again, the digit in position 0 being least significant). For instance, given $[1, 1, 1, 0, 1]$ on input (our representation of the binary number 10111), we should get $[3, 2]$ on output (our representation of the decimal number 23).

```python
function bin2dec(v)
Input: An n-bit integer $v \geq 0$ (binary digits)
Output: The vector of decimal digits of $v$
if $v = []$: return $w = []$
$z = \text{bin2dec}([v/2])$
$w = \text{By2inDec}(z)$
if $v$ is odd:
  if $w = []$: $w = [1]$
  else: $w[0] = w[0] + 1$
return $w$
```

3This is just a convention. We could equally well choose a different representation, with the rightmost digit being the least significant one (this is how we typically write numbers on paper). Under this schema, 6 would be represented as $[1, 1, 0]$. No matter which representation we choose, we have to stay consistent.
An iterative version of the algorithm is now easy to derive:

\[
\text{function ItBin2dec}(v) \\
\text{Input: } \text{An } n\text{-bit integer } v \geq 0 \text{ (binary digits)} \\
\text{Output: } \text{The vector } w \text{ of decimal digits of } v
\]

\[
w = [] \\
\text{for } i = \text{size}(v) - 1 \text{ downto } 0:\ \\
\quad w = \text{By2inDec}(w) \\
\quad \text{if } v[i] \text{ is odd:} \\
\quad \quad \text{if } w = []: \quad w = [1] \\
\quad \quad \text{else: } \quad w[0] = w[0] + 1 \\
\text{return } w
\]

The algorithm \text{By2inDec}(z) takes as input the vector of decimal digits of an integer \(z\) and returns the vector of decimal digits of \(2z\).

Can you design two similar algorithms for converting from decimal to binary? Can you generalize and propose algorithms that change representations in base 2 to representations in any given base \(b > 2\)? How to convert a representation in any given base \(b > 2\) to base 2?

### 2.1 Addition

The following simple property underlies algorithms for adding two numbers in all position systems with base \(b \geq 2\):

The sum of any three digits is at most two digits long

The less significant of the two digits serves as the next digit of the sum, the other digit (possibly 0) serves as the carry which, due to that property is always a single digit.

Thus, the addition algorithm takes \(O(n)\) steps to add two \(n\)-bit binary numbers. (Again, to add two 32-bit long numbers, a constant number of computer steps are needed; but we will be adding much longer numbers.)

Is there a faster algorithm? No. Just to read the numbers in takes \(\Omega(n)\) steps.

However, it is a good idea to utilize the size of computer words. Let us use an array \(a[0..n - 1]\) for storing integers, and let us assume that each entry can store up to \(k\) bits. Then, we can store \(kn\) bits in this array. To add two sequences of bits stored in this way in arrays \(a\) and \(b\), one can think of that task as adding two numbers in base \(b_k = 2^k\) (with digits 0, 1, \ldots, \(2^k - 1\), just as large as can fit). The basic addition algorithm is the same as that for base 2 or base 10.

We also note that binary sequences so represented can easily be multiplied by 2, and divided by two as well (shifted to the left and to the right). For instance, to shift to the left, let us assume that we get a bit \(c\) from the right (initially, \(c = 0\)). Let the content of the
current cell be \( a \). The bit to send to the left is set to \( c' = \lfloor a/b_k-1 \rfloor \), and the new content of the current cell is set to \( a' = (2 \times (a \mod b_k-1)) + c \). To shift to the right, let \( c \) stand for the bit to be sent to the right, initially \( c = 0 \). Let \( a \) be the content of the current cell. Then, \( c' = a \mod 2 \), and \( a' = c \times b_k + \lfloor a/2 \rfloor \).

### 2.2 Subtraction

We describe the algorithm for computing \( x - y \) under the assumption that \( x \geq y \geq 0 \). We also assume that positive integers are represented in a straightforward way by a sequence of their significant binary digits (vectors of digits). The algorithm is based on the idea of guessing the digits of the difference. There are better algorithms (for instance, under the two’s complement representation) but this one will do for us.

```python
function subtract(x, y)
    Input: Two \( n \)-bit integers \( x \) and \( y \), where \( x \geq y \geq 0 \),
    Output: Their difference \( d \)
    pad \( y \) with leading 0s so that \( size(y) = size(x) \)
    \( c = 0 \)
    for \( i = 0 \) to \( n - 1 \):
        if \( c + x[i] + y[i] \) is odd: \( d[i] = 1 \)
        else: \( d[i] = 0 \)
        if \( c + y[i] + d[i] \geq 2 \): \( c = 1 \)
        else: \( c = 0 \)
    remove leading 0s from \( d \) (\( d \) may get empty)
    return \( d \)
```

### 2.3 Multiplication and division

Multiplication of two \( n \)-bit numbers is accomplished by a series of additions. Numbers in these additions are no more than \( 2n-1 \)-bit long and the result is no more than \( 2n \)-bits long. The numbers to be added are obtained by shifting the first number (if they are not of even length, we should choose a longer number to be the “first” number). The second number defines the shifts (or how many 0s to add at the end).

\[
\begin{array}{cccc}
1 & 1 & 0 & 1 \\
\times & 1 & 0 & 1 \\
\end{array}
\quad \text{(binary 13)}
\]

\[
\begin{array}{cccc}
1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 \\
+ & 1 & 1 & 0 & 1 \\
\end{array}
\quad \text{(binary 143)}
\]
Since shifts and additions can each be accomplished in time $O(n)$, the total time needed by that multiplication algorithm is $O(n^2)$.

Can we do better? Already Al Khwarizmi new another method (it is still in use in some places around the world). It is based on the following recursive formula:

$$x \cdot y = \begin{cases} 2(x \lfloor y/2 \rfloor) & \text{if } y \text{ is even} \\ x + 2(x \lfloor y/2 \rfloor) & \text{if } y \text{ is odd} \end{cases}$$

To run it, all you need to do is to know how to multiply and divide by 2 and how to add. Here is an illustration

$$73 \times 35$$
$$146 \times 17 + 73$$
$$292 \times 8 + 73 + 146$$
$$584 \times 4 + 73 + 146$$
$$1168 \times 2 + 73 + 146$$
$$2336 \times 1 + 73 + 146$$

To get the answer, remove rows where the second number is even and add up first numbers in the remaining rows.

$$73 \times 35$$
$$146 \times 17 + 73$$
$$2336 \times 1 + 73 + 146$$

= 2555

In the case when $x$ and $y$ are given in binary, it is especially easy to implement, as multiplication and division of binary numbers by 2 is easy (shifts).

Now the algorithm.

```python
def multiply(x, y):
    Input: Two integers $x$ and $y$, where $x, y \geq 0$
    Output: Their product

    if $y = 0$: return 0
    $z = multiply(x, \lfloor y/2 \rfloor)$
    if $y$ is even:
        return $2z$
    else:
        return $x + 2z$
```

12
If \( n = \max\{\text{size}(x), \text{size}(y)\} \), the algorithm terminates after \( n \) recursive calls (each call halves \( y \), that is, the length of \( y \) decreases by 1 bit). Each recursive call requires a test for equality to 0; a division by 2 (right shift); a test for odd/even (last bit look-up); a multiplication by 2 (left shift); possibly an addition — for the total of \( O(n) \) steps. Thus, the total time is \( O(n^2) \), as before.

An iterative version of that algorithm is more efficient (both in terms of time and memory).

```plaintext
function it-multiply(x, y)
Input: Two integers \( x \) and \( y \), where \( x, y \geq 0 \)
Output: Their product
\( z = 0 \)
for \( i = \text{size}(y) - 1 \) downto 0:
    \( z = 2z \)
    if \( y[i] = 1 \): \( z = z + x \)
return \( z \)
```

### 2.4 Division

A similar idea works. Say, we are to divide \( x \) by \( y \) (\( y \geq 1 \); if \( y < 0 \), divide \( -x \) by \( -y \)). That means we are to find a quotient \( q \) and a remainder \( r \) such that \( x = yq + r \) and \( 0 \leq r < y \).

```plaintext
function divide(x, y)
I: Two integers \( x \) and \( y \), where \( x \geq 0, y \geq 1 \)
O: The quotient and the remainder of \( x \) divided by \( y \)
if \( x = 0 \): return \((q, r) = (0, 0)\)
\((q, r) = \text{divide}([x/2], y)\)
\( q = 2q, r = 2r \)
if \( x \) is odd: \( r = r + 1 \)
if \( r \geq y \): \( r = r - y, q = q + 1 \)
```

When we compute \((q, r)\) in the second line of the algorithm, we have the identity (\( q \) is the quotient and \( r \) is the remainder for the division of \([x/2]\) by \( y\))

\[
[x/2] = y \cdot q + r,
\]

which implies

\[
2 \cdot [x/2] = y \cdot 2q + 2r,
\]

We also note that \( r < y \) and so, \( 2r + 1 < 2y \). Thus, if \( 2r + 1 \geq y \), then \( 0 \leq 2r + 1 - y < y \).

The algorithm works in \( O(n^2) \) time, where \( n = \max\{\text{size}(x), \text{size}(y)\} \).

As before, we present also a non-recursive version of the division algorithm.
function it-divide\((x, y)\)
I: Two integers \(x\) and \(y\), where \(x \geq 0\), \(y > 0\)
O: The quotient and the remainder of \(x\) divided by \(y\)

\(q = 0\)
\(r = 0\)
for \(i = \text{size}(x) - 1\) downto 0:
\(q = 2q\)
\(r = 2r\)
if \(x[i] = 1\):
\(r = r + 1\)
if \(r \geq y\):
\(r = r - y\)
\(q = q + 1\)
return \((q, r)\)

2.5 Modular arithmetic

A number system determined by an integer \(N \geq 2\). It deals with the set \(\mathbb{Z}_n = \{0, 1, \ldots, N - 1\}\). This set represents all possible remainders when dividing by \(N\). One can view it as modeling all \(N\) classes of numbers with the same behavior relative to division by \(N\). Each class consists of all numbers that are pairwise congruent modulo \(N\):

\[ x \equiv y \pmod{N} \quad \text{if and only if} \quad N \text{ divides } x - y. \]

All arithmetic operations, \(+, -, \cdot, /\), as well as exponentiation can be defined in that set. The result of a modular operation is defined as the remainder of the division by \(N\) of the result of the corresponding regular operation. The modular addition and multiplication satisfy the associativity and commutativity. In addition, the distributivity of multiplication wrt addition holds, too.

**Substitution rule:** If \(x \equiv x' \pmod{N}\) and \(y \equiv y' \pmod{N}\), then

\[ x + y \equiv x' + y' \pmod{N} \quad \text{and} \quad x \cdot y \equiv x' \cdot y' \pmod{N} \]

That implies that

\[ x^y \equiv x'^y \pmod{N}. \]

Application: compute \(2^{345} \pmod{31}\) and \(2^{345} \pmod{30}\). Later, we will present a modular exponentiation algorithm that will also be applicable to such problems (although tedious to use if exponents are large).

2.5.1 Modular addition and multiplication

To add two numbers \(x\) and \(y\) modulo \(N\), we perform the regular addition (the result is between 0 and \(2(N - 1)\)). If the result exceeds \(N - 1\), we subtract \(N\). The overall computation
consists of one addition, one comparison and (possibly) one subtraction. Thus, modular addition runs in $O(n)$ time (recall: $n$ in this section always stands for the length of the binary representation of numbers).

To multiply two numbers $x$ and $y$ modulo $N$, we perform the regular multiplication (the result is between 0 and $(N - 1)^2$ and requires no more than $2n$ bits. We then compute the remainder of the division of $x \cdot y$ by $N$ using the quadratic time division algorithm. Thus, modular multiplication remains a quadratic procedure.

The division is tricky. Unlike in the real division, where only the division by 0 is illegal here there may be more exceptional cases (e.g., try to divide 3 by 3 modulo 6; is it 1 or is it 3 or is it 5?). However, we will show later how to do modular division, whenever it is legal, in cubic time.

### 2.5.2 Modular exponentiation

The goal is to compute $x^y \mod N$. The brute-force approach is to compute $x^y$ and then to compute the remainder of the division of the result by $N$. The problem is that $x^y$ may have huge length. Say, $x$ and $y$ are “proper” 30-bit numbers, that is each is at least $2^{29}$. The length of the representation of $x^y$ is

$$\lceil \log(x^y + 1) \rceil \geq \log x^y = y \log x \geq 2^{29} \cdot 29 \geq 15 \cdot 10^9,$$

that is, more than 15 billion bits long.

Fortunately, we can use the substitution rule. One way to go is below:

$$x \mod N \to x^2 \mod N \to x^3 \mod N \ldots \to x^y \mod N.$$ 

We start with $x \mod N$ and at each step we multiply the current value by $x$ modulo $N$. The final and all intermediate results do not exceed $N - 1$ and so can be represented with $n$ bits. The total time is $O(y \cdot n^2)$. But the presence of $y$ in the formula still a problem.

However, there is a better way based on squaring:

$$x \mod N \to x^2 \mod N \to x^4 \mod N \ldots \to x^{2^k} \mod N.$$

That requires only $k$ squarings to compute $x^{2^k} \mod N$.

Assuming $y$ is $\langle a_{n-1}, \ldots, a_0 \rangle$ in binary,

$$x^y = x^{a_{n-1}2^{n-1}+\ldots+a_12^1+a_02^0} = x^{a_{n-1}2^{n-1}} \cdot \ldots \cdot x^{a_12^1} \cdot x^{a_02^0}$$

Thus, to compute $x^y \mod N$, we need to compute each factor, requiring $n - 1$ squarings modulo $N$, for each of the largest of them, with all smaller ones needed computed along the way. Then we need to multiply these (at most) $n$ factors (modulo $N$), which takes another $n - 1$ multiplications modulo $N$. Thus, modular exponentiation can be accomplished in time $O(n^3)$. 

15
To package this idea differently, we have the following formula:

\[
x^y = \begin{cases} 
(x^{\lfloor y/2 \rfloor})^2 & \text{if } y \text{ is even} \\
x \cdot (x^{\lfloor y/2 \rfloor})^2 & \text{if } y \text{ is odd}
\end{cases}
\]

It gives rise to a recursive algorithm:

```plaintext
function modexp(x, y, N)
Input: Three integers x, y and N, 0 ≤ x, y ≤ N − 1
Output: x^y mod N

if y = 0: return 1
z = modexp(x, ⌊y/2⌋, N)
if y is even:
    return z^2 mod N
else:
    return x \cdot z^2 mod N
```

And its non-recursive version:

```plaintext
function it-modexp(x, y, N)
Input: Three integers x, y and N, 0 ≤ x, y ≤ N − 1
Output: x^y mod N

z = 1
for i = size(y) − 1 downto 0:
    z = z^2 mod N
    if y[i] = 1: z = zx mod N
return z
```

The algorithms run in time \(O(n^3)\), where \(n = size(N)\).

For instance, to compute \(9^{1321} \mod 14\), we first find the binary representation of 1321: 10100101001. Therefore, the subsequent values of \(z\) are (here \(n = 11\))

\[
\begin{align*}
z &= 1 \text{ (initial assignment)} \\
i &= 10: z = (1^2 \mod 14) \times 9 \mod 14 = 9 \\
i &= 9: z = 9^2 \mod 14 = 11 \\
i &= 8: z = (11^2 \mod 14) \times 9 \mod 14 = 9 \times 9 \mod 14 = 11 \\
i &= 7: z = 11^2 \mod 14 = 9 \\
i &= 6: z = 9^2 \mod 14 = 11 \\
i &= 5: z = (11^2 \mod 14) \times 9 \mod 14 = 9 \times 9 \mod 14 = 11 \\
i &= 4: z = 11^2 \mod 14 = 9 \\
i &= 3: z = (9^2 \mod 14) \times 9 \mod 14 = 11 \times 9 \mod 14 = 1 \\
i &= 2: z = 1^2 \mod 14 = 1 \\
i &= 1: z = 1^2 \mod 14 = 1 \\
i &= 1: z = (1^2 \mod 14) \times 9 \mod 14 = 9
\end{align*}
\]
2.5.3 Euclid’s algorithm for GCD

Euclid’s rule. If \( x \) and \( y \) are positive integers and \( x \geq y \), then \( \gcd(x, y) = \gcd(x - y, y) \) and \( \gcd(x, y) = \gcd(x \mod y, y) \).

The second part of the assertion follows from the first one. The first one is evident.

The rule implies the algorithm also attributed to Euclid.

Euclid of Alexandria, 325 - 265 BCE

function Euclid\(_x\,y\) 
Input: Two integers \( x, y \) with \( x \geq y \geq 0 \)
Output: \( \gcd(x, y) \) (assumes that \( \gcd(0, 0) = 0 \))

\[
\begin{align*}
\text{if } y = 0 & : \text{ return } x \\
\text{return } & \text{Euclid}(y, x \mod y)
\end{align*}
\]

The corresponding non-recursive algorithm follows:

\[
\begin{align*}
\text{function it-Euclid}\_x\,y \\
\text{Input: Two integers } x, y \text{ with } x \geq y \geq 0 \\
\text{Output: } \gcd(x, y) \text{ (assumes that } \gcd(0, 0) = 0 \text{)}
\end{align*}
\]

\[
\begin{align*}
\text{while } y > 0 : \\
\quad z & = x \mod y \\
\quad x & = y \\
\quad y & = z \\
\text{return } & x
\end{align*}
\]

Lemma 2.1 If \( x \geq y > 0 \), then \( x \mod y < x/2 \).

It follows that in every two consecutive recursive calls \( x \) and \( y \) are at least halved, that is, their binary representations decrease by at least one bit. Thus, the algorithm terminates within at most \( 2n \) recursive calls (where \( n = \max\{\text{size}(x)\} \)). Each call involves division (quadratic time). Thus, Euclid’s argument runs in time \( O(n^3) \). Similarly, we argue that in the iterative Euclid algorithm there are no more than \( 2n \) iterations of the while loop and we get the same time estimate.

2.5.4 An important extension

Lemma 2.2 (Bezout Property) Let \( z > 0 \). Then, \( z \) divides both \( x \) and \( y \) and \( z = ax + by \) for some integers \( a \) and \( b \) if and only if \( z = \gcd(x, y) \).

Proof: \((\Rightarrow)\) By the assumption, \( z \leq \gcd(x, y) \). On the other hand, \( \gcd(x, y) \) divides \( z \). Thus, \( \gcd(x, y) \leq z \) and the assertion follows.

\((\Leftarrow)\) It is clear that \( z \) divides \( x \) and \( y \). The existence of the numbers \( a \) and \( b \) follows from the correctness of the construction below. \( \square \)
Before we describe the algorithm, we illustrate the method of finding such numbers. Let us assume that \( x \geq y \). The process consists of constructing a sequence of integers starting with \( x \) and \( y \), with each next number \( z \) in the sequence equal to \( z'' \mod z' \), where \( z' \) is the previous number and \( z'' \) is the one preceding \( z' \). That sequence is constructed exactly as in the Euclid’s algorithm and ends with the greatest common divisor for \( x \) and \( y \) (we know it is time to stop when the next number is 0).

In addition, for each element \( z \) in that sequence, we construct pairs \((a, b)\) of integers such that \( z = ax + by \). The last pair is what is needed to complete the proof above. The first two pairs are set to \((1, 0)\) and \((0, 1)\). They clearly have the desired property. For the general step, let us denote the two most recent pairs by \((a'', b'')\) and \((a', b')\) (with the latter being the most recent one). Let \( z = z'' \mod z' \) be the next number in the sequence produced by the Euclid algorithm (iterative version) and let \( q = z'' / z' \). Then \((a, b)\) can be computed as follows:

\[
\begin{align*}
    a &= a'' - qa' \\
    b &= b'' - qb'.
\end{align*}
\]

This implies the following algorithm:

```plaintext
function it-extended-Euclid(x, y)
Input: Two integers \( x, y \) with \( x \geq y \geq 0 \)
Output: integers \( a, b, z \) such that \( z = \gcd(x, y) \) and \( z = ax + by \)

\((a'', b'') = (1, 0)\)
\((a', b') = (0, 1)\)
while \( y > 0 \):
    \( q = x / y \)
    \( a = a'' - qa' \)
    \( b = b'' - qb' \)
    \((a'', b'') = (a', b')\)
    \((a', b') = (a, b)\)
    \( z = x \mod y \)
    \( x = y \)
    \( y = z \)
return \((a'', b'', x)\)
```

There is also a recursive version of that algorithm.

```plaintext
function extended-Euclid(x, y)
Input: Two integers \( x, y \) with \( x \geq y \geq 0 \)
Output: integers \( a, b, z \) such that \( z = \gcd(x, y) \) and \( z = ax + by \)

if \( y = 0 \): return \((1, 0, x)\)
\((a', b', z) = \text{extended-Euclid}(y, x \mod y)\)
return \((b', a' - \lfloor x / y \rfloor b', z)\)
```
The correctness is guaranteed by the following lemma.

**Lemma 2.3** For any positive integers \( x \) and \( y \), the extended Euclid algorithm returns integers \( a \), \( b \) and \( z \) such that \( z = \gcd(x, y) \) and \( z = ax + by \).

Proof: Induction on \( y \). The base of the induction easy to verify. To show the assertion for \( y > 0 \), we note that \( x \mod y < y \). Thus, by the inductive hypothesis, \( z = \gcd(y, x \mod y) \) and

\[
z = a'y + b'(x \mod y)
\]

By the Euclid’s rule, \( z = \gcd(x, y) \). Let \( x = qy + r \), where \( 0 \leq r < y \). Clearly, \( q = \lfloor x/y \rfloor \). Thus,

\[
z = b'r + a'y = b'(x - \lfloor x/y \rfloor y) + a'y = b'x + (a' - \lfloor x/y \rfloor b')y,
\]

as required by the inductive step. \( \square \)

The running time analysis is the same as for the Euclid’s algorithm as far as the depth of the recursion goes. Moreover, one can show that all numbers \( a, b \) produced in the course of the (non-recursive) are \( n \)-bit numbers. Thus, the entire analysis carries over.

### 2.5.5 Modular division

We are ready now to show a method for modular division. We say that \( 0 \leq x \leq N - 1 \) is the multiplicative inverse of \( a \) modulo \( N \) if \( ax \equiv 1 \pmod{N} \). Modular division can be defined in terms of multiplicative inverses.

In real arithmetic, every real other than 0 has exactly one inverse. Here, any integer \( a \) has at most one multiplicative inverse modulo \( N \), but some non-zero integers may have no multiplicative inverse.

For instance, for every \( i \), \( 2 \cdot i \equiv 0, 2, \text{ or } 4 \pmod{6} \). Thus, 2 has no multiplicative inverse modulo 6. In general, if \( \gcd(a, N) > 1 \), \( a \) has no multiplicative inverse modulo \( N \) (\( a \equiv aq \pmod{N} \), for some \( q \); thus, \( \gcd(a, N) \) divides \( aq \pmod{N} \)).

How to show that two different multiplicative inverses of \( a \) are impossible? Let us assume that \( ax \equiv 1 \pmod{N} \) and \( ax' \equiv 1 \pmod{N} \) for some \( 0 \leq x < x' \leq N - 1 \). Thus, \( a(x' - x) \equiv 0 \pmod{N} \). Since inverse exists, \( \gcd(a, N) = 1 \). Thus, \( N \) divides \( x' - x \). As \( x' - x < N \), \( x = x' \).

Finally, if \( \gcd(a, N) = 1 \), does there exist the multiplicative inverse of \( a \) modulo \( N \)? The answer is Yes! Indeed, we have integers \( x \) and \( y \) such that

\[
1 = xa + yN.
\]

Thus, \( xa \equiv 1 \pmod{N} \) and \( a \) has a multiplicative inverse. One can use the extended Euclid algorithm to compute \( x \) and \( y \) such that \( 1 = yN + xa \) and \( 0 \leq x \leq N - 1 \) (can be computed in time \( O(n^3) \)).
2.6 Primality testing

This is another basic task essential for cryptographic applications of algorithms with numbers we are discussing here. Here is the key result.

**Theorem 2.4 (Fermat’s little theorem)** If \( p \) is prime, then for every \( a \) such that \( 1 \leq a < p \),

\[
a^{p-1} \equiv 1 \pmod{p}.
\]

Proof: The numbers \( a \cdot i \mod p, i = 1, 2, \ldots, p-1 \) are all distinct. Indeed, modular division by \( a \) is legal. Thus, if \( a \cdot i \equiv a \cdot j \pmod{p} \), then dividing both sides by \( a \) yields \( i \equiv j \pmod{p} \).

Thus,

\[
\{1, 2, \ldots, p-1\} = \{a \cdot i \mod p \mid i = 1, 2, \ldots, p-1\}.
\]

Thus, the products of all elements in the two sets (modulo \( p \)) are the same:

\[
(p-1)! \equiv a^{p-1} \cdot (p-1)! \pmod{p}
\]

Dividing by \((p-1)!\), which we can do as it is relatively prime with \( p \), yields the result. \( \square \)

**Corollary 2.5** If \( p \) is prime, then for every \( a \) such that \( a \mod p \neq 0 \),

\[
a^{p-1} \equiv 1 \pmod{p}.
\]

Proof: It follows that \( a = kp + b \), where \( 1 \leq b < p \). Clearly,

\[
a^{p-1} \equiv b^{p-1} \pmod{p}
\]

and the result follows by Fermat’s little theorem. \( \square \)

We now propose the first (highly imprecise, as we will see) method to test primality.

```plaintext
function primality(N)
Input: Positive odd integer \( N \geq 2 \)
Output: yes/no
Pick a positive integer \( a < N \) at random
if \( a^{N-1} \equiv 1 \pmod{N} \):
    return yes
else:
    return no
```

The problem is that this algorithm may output “yes” for non-prime number (it will always correctly output “yes” if \( N \) is prime, by Fermat’s little theorem). But, surprisingly, this algorithm works well if \( a = 2 \) (as well as for several other values of \( a \)) and we apply it to a randomly chosen integer. For instance, when applied to a randomly chosen 512-bit integer, the probability of an error is less than \( 10^{-20} \). We can say nothing, though, about the correctness when that algorithm is applied to a concrete input.

We will now improve on that algorithm.
function primality2(N)
Input: Positive odd integer $N \geq 2$
Output: yes/no

Pick positive integers $a_1, \ldots, a_k < N$ at random
if not composite$(a_i, N)$ for all $i = 1, \ldots, k$:
    return yes % Almost surely prime
else:
    return no % Definitely composite

The procedure composite$(a, N)$ is meant to decide whether $a$ demonstrates that $N$ is composite. It correctly returns “true” and almost always correctly returns “false.” It is slightly more involved than the simple test $a^{N-1} \equiv 1 \pmod N$ we were using before. It is due to Miller and Rabin (the latter a 1976 Turing Award winner for his work on non-determinism, jointly with Dana Scott). Its description follows.

function composite$(a, N)$
Input: Positive odd integer $N \geq 2$, integer $a$, $1 < a < N$
Output: yes/no
find $t$ and $u$ such that $N-1 = 2^t u$, with $t \geq 1$ and $u$ odd
$x_0 = a^u \pmod N$
for $i = 1, \ldots, t$
    $x_i = x_{i-1}^2 \pmod N$
    if $x_i = 1$ and $x_{i-1} \neq 1$ and $x_{i-1} \neq N-1$:
        return true
    if $x_t \neq 1$:
        return true
return false

To argue its “almost” correctness, we note that $x_t = a^{2^t u} = a^{N-1} \pmod N$. Now, if the algorithm returns true from the first return command, we have just found an integer $c$ such that $c \neq 1$, $c \neq N-1$ and $c^2 \equiv 1 \pmod N$, that is, the so called non-trivial square root of 1 modulo $N$. This is only possible if $N$ is composite. If the algorithm returns true from the second return command, it is again correct (by the Fermat’s theorem, $N$ cannot be prime).

The next lemma shows that the probability that composite is in error is low.

Lemma 2.6 If $N$ is odd and composite, the number of $a$’s such that $1 < a < N$ and composite$(a, N)$ returns true is at least $3(N-1)/4$

Thus, when $N$ is odd and composite, the probability of getting false from composite$(a, N)$ (suggesting incorrectly that $N$ is prime) is at most $\frac{1}{2^k}$. Consequently, the probability of an error by the algorithm primality2 is bounded from above by $1/2^{2k}$. Indeed, we must have that $N$ divides $x^2 - 1 = (x-1)(x+1)$. Since $1 < x < N-1$ and $N$ is a prime, $N$ divides $x-1$ or $N$ divides $x+1$. In the first case, it follows that $x = 1$, in the second case, it follows that $x = N-1$. 

21
2.7 Generating primes

**Theorem 2.7 (Lagrange’s prime number theorem)** Let \( \pi(x) \) be the number of primes \( \leq x \). Then \( \pi(x) \approx x/(\ln x) \), or more precisely

\[
\lim_{x \to \infty} \frac{\pi(x)}{x/(\ln x)} = 1.
\]

Thus, prime numbers abound! If \( x \) is a large number, say with \( n \) bits, the probability that a number selected from \( \{0, 1, 2, \ldots, x\} \) uniformly at random is prime is about

\[
\frac{1}{\ln x} = \frac{1}{n}.
\]

(Recall that \( \log_2 e \approx 1.443 \).)

It is therefore simple to generate primes:

- Pick an \( n \)-bit integer \( N \)
- Run a primality test on \( N \)
- If it passes the test, output \( N \), otherwise repeat the process

How fast is the algorithm? The probability that \( N \) is prime is about \( 1.443/n \) (by the theorem). If it is, the algorithm terminates (as prime numbers are guaranteed to pass the primality test). Thus, the algorithm will halt in the first iteration with the probability \( 1.443/n \), and one can show that the expected number of iterations to first success (inclusive) is \( n/1.443 \) (cf. Bernoulli process).

What primality test to use? Since the numbers the test is given are randomly generated (and not given by an adversary), the primality test based on the Fermat’s test with \( a = 2 \) (or with \( a = 2, 3 \) and 5 for smaller error) might be good enough (cf comments above; here we add that this simple algorithm errs with probability less than \( 10^{41} \) on 1024-bit numbers).

2.8 Cryptography

We will now apply all these results to present the Rivest-Shamir-Adelman (RSA) cryptosystem.

The setting includes Alice and Bob, who wish to communicate in private, and Eve — an eavesdropper. Alice wants to sent to Bob a specific message, say a binary string \( x \). She first encodes \( x \) into another binary string \( e(x) \), using an *encryption* function \( e \). Next, she sends \( e(x) \) to Bob. Bob applies to the string received the *decryption* function \( d \) and reads of \( x \) as \( d(e(x)) \).

Eve might intercept \( e(x) \). But the encryption function should be such that without knowing \( d \), Eve cannot do anything with the information she picked up.

This is a simple schema used for centuries. It required that Alice and Bob meet beforehand and together choose \( e \) and \( d \), and that later \( d \) and \( e \) be kept secret. Eve’s only hope is to collect several messages and use them to partially figure out the codebook.
2.8.1 Private-key schemes

A nice theoretical insight into private-key systems is given by the one-time pad method. Alice and Bob meet and choose a binary string $r$ of the same length, say $n$ (which may be very long), as the message Alice will later send to Bob. The encryption and decryption functions are defined as follows:

$$e_r(x) = d_r(x) = x \oplus r.$$ 

How $r$ should be selected? Selecting each bit at random by flipping a coin. That way Eve, having intercepted $x \oplus r$ has no information on which of the possible $2^n$ strings was sent by Alice, to say it precisely, all are equally likely.

But after using $r$, it has to be discarded and another string used (hence the name). If Alice uses it twice to send $x$ and $y$, and Eve intercepts $x \oplus r$ and $y \oplus r$, then Eve can recover $x \oplus y$, which may give her important clues (for instance, if one message contains a long sequence of 0s, the corresponding fragment from the second message is revealed).

At the other end of the spectrum is the AES (advanced encryption standard), approved in 2001 by the US National Institute of Standards and Technologies (NIST). AES require that Alice and Bob agree on a private key of length, say 128. This key defines the functions $e_r$ and $d_r$, which are bijections from 128-bit strings to 128-bit strings (the description is public; depending on the number of the “rounds” used the performance of the system deteriorates but its security grows).

To send a message $x$, Alice breaks it into 128-bit chunks, encodes each using $e_r$, and sends the encoded strings to Bob. The security of that schema has not been rigorously established. But at present general public does not know how to break it.

What to do if Alice and Bob have no way to meet beforehand? The answer is in public-key cryptography. It allows Alice and Bob to share a private key (without having to meet) and use it with private-key systems such as AES. It is here that our number algorithms will be used.

2.8.2 RSA — an example of a PKC

In public-key cryptosystem, everybody who wishes to receive secret messages publishes their public key. Thus, to send a message $x$ to Bob, Alice uses Bob’s public key to encode $x$. Bob uses his private key(not shared with anybody) to decode it.

RSA is an example of a public-key cryptosystem. It is based on the following property.

**Theorem 2.8** Let $p$ and $q$ be two primes such that $p \neq q$ and $N = pq$. For every $e < N$ relatively prime to $(p - 1)(q - 1)$:

1. The mapping $x \mapsto x^e \mod N$ is a bijection on $\{0, 1, \ldots, N - 1\}$

2. The inverse mapping is given by $y \mapsto y^d \mod N$, where $d$ is the inverse of $e$ modulo $(p - 1)(q - 1)$.
Proof: (2) implies (1). Thus, we will prove (2) only. Specifically, we need to prove that 

\[(x^e)^d \equiv x \mod N.\]

We first prove that \(x^{ed} \equiv x \mod p\). It clearly holds for every \(x \equiv 0 \mod p\). Thus, let \(x \not\equiv 0 \mod p\). Then, \(x^{p-1} \equiv 1 \mod p\) (by the corollary to Fermat’s little theorem). Since \(ed \equiv 1 \mod (p-1)(q-1)\), \(ed = 1 + k(p-1)(q-1)\). Thus, \(x^{ed} \equiv x^{k(p-1)(q-1)} \equiv x^{[x^{p-1}]^{k(q-1)}} \equiv x \mod p\), as needed. Similarly, \(x^{ed} \equiv x \mod q\).

It follows that \(x^{ed} - x\) is divisible by \(p\) and by \(q\). Since \(p\) and \(q\) are prime, \(x^{ed} - x\) is divisible by \(N\). □

The RSA system can now be described as follows:

**Bob chooses his public and secret keys**

- He picks two large \(n\)-bit random primes \(p\) and \(q\)
- His public key is \((N, e)\), where \(N = pq\) and \(e\) is a \(2n\)-bit number relatively prime to \((p-1)(q-1)\)
- His secret key is \(d\), the inverse of \(e\) modulo \((p-1)(q-1)\), computed using the extended Euclid’s algorithm.

**Alice sends a message to Bob**

- She looks up his public key \((N, e)\) and sends him \(y = (x^e \mod N)\) computed using an efficient algorithm for modular exponentiation
- He decodes the message by computing \(y^d \mod N\) (again, using the modular exponentiation algorithm).

How secure is this schema? Its security hinges on an assumption:

Given \(N, e\) and \(y = x^e \mod N\), it is computationally intractable to determine \(x\).

The assumption is quite plausible. To get \(x\), Eve could use guessing (little chance of success), or she could try factoring \(N\). But factoring is believed to be hard (no fast factoring methods are known).

Finally, we note that RSA is relatively slow (encoding and decoding depend on modular exponentiation, which has cubic running time). Thus, RSA is used to communicate a private key. Once that is done, AES (or some other private-key system) is used.

Here is an example. Let us pick \(p = 5\) and \(q = 11\). Then \(N = 55\). We can choose \(e = 3\) as \(\gcd(40, 3) = 1\). We now compute \(d = 27\) as the multiplicative inverse of \(3\) modulo 40. Since 55 = 110111 in binary, all 5-bit strings of 0s and 1s (the corresponding integers, to be precise) fall in the range \(\{0, \ldots, N-1\}\) and can be encoded and then decoded in the RAS system based on our selections. Of course some 6-bit strings can too (those that represent numbers smaller than 55, for instance, 100011 but not 111000).

Given a 5-bit string, say \(x = 11010\) (representing \(x = 26\)), to encode it, we compute \(x^e \mod 55\). This turns out to be 31, that is, 11111 in binary. Some encrypted strings will have 6-bits.
2.9 Hashing

This is a method for maintaining an ever-changing list (set, database) of records, with access, insert and deletion as the three basic operations. Structures of that sort (sets with the three operations mentioned) are called dictionaries.

The records are identified by keys (say IP addresses of clients of an on-line business, UK IDs of students at the University of Kentucky). The space of keys is huge in comparison with the number of records to be maintained in the set. In general, we will assume that keys are integers from the set \([0..N-1]\) for some large integer \(N\). Typically, \(N\) is a power of 10 (\(10^9\) social security numbers, \(10^8\) UK IDs), 2 (\(2^{32}\) IP addresses), 26 (26 strings of length 10 over the English alphabet). We will write \(M\) for the integer representing a limit on the number of records (elements) we must be prepared to handle.

As we said, the list of elements will be assumed not to exceed some rather small number (say around 250, for the on-line business example). In that example, it does not make sense to have an array of \(2^{32}\) entries labeled by IP addresses, with most entries being null and only at most 250 containing an active customer record. While access, insert and delete times are excellent, the approach is impractical as it wastes space.

We could use a linked list of the customer records. Then, there would be no waste of space. But access times would be unacceptable (the whole list might have to be inspected).

Hash tables provide an answer. These tables are not labeled by all possible record ID’s. Instead, each ID \(x\) gets a nickname, \(h(x)\) (\(h\) for the “hash” function) from some much smaller set (say \([0..256]\); why that range and not \([0..249]\) in a moment). A record with ID \(x\) is stored in the entry labeled with the nickname \(h(x)\) of \(x\). Records are arranged into linked lists if multiple records are to be stored in the same entry of the array (a distinct possibility, as many ID’s by necessity have the same nickname). However, if we can ensure that this does not happen often, we get a very efficient way to maintain our set of records (an array of very short lists).

In our example, our hash function could assign the first segment of an IP address as its nickname. The range \([0..255]\) is within the range \([0..256]\) we designated for nicknames. So, one entry in the array will not be used for sure and, possibly, some others will not be used either (if some integers in \([0..255]\) do not show up as the first segments of the IP addresses of our customers). That is okay. The array is small anyway and if some entries are empty it is not a problem.

However, the function we just proposed is vulnerable to adverse selections of IP addresses. It works well if the IP addresses of our customers form a random sample of all addresses (under the uniform distribution). But we cannot be certain that is the case (there is a structure in IP addresses and if all our customers are from one or two broad geographic regions, there will be very many repetitions).

What we want is a randomly selected hash function, our guarantee that it will work well coming not from any assumptions about the space of ID’s it will be applied to (we may have no control over that), but from the properties of the family of hash functions we draw hash functions from.
Here is a simple family we might use in our IP address example. We select a prime that encompasses the range of nickname values (the required range of the hash function). In our case, we could select N=257 (which is prime). And, indeed, that is what we did select. From now on this prime represents the range of the hash function and so, we will write $M$ for it. We view IP addresses (in general, ID’s) as 4-tuples $(x_0, \ldots, x_3)$ with $0 \leq x_i \leq 256$ (some of these tuples are not valid IP address, but that is okay; important thing is that every IP address is of that form). We now consider the space of hash functions determined by 4-tuples $a = (a_0, \ldots, a_3)$ with $0 \leq a_i \leq 256$ as follows:

$$h_a(x_0, \ldots, x_3) = \sum_{i=0}^{3} a_i \cdot x_i \mod 256$$

If these coefficients are drawn at random, our hash function is good in the following sense (the lemma is formulated for an arbitrary prime $M$, in the example we are discussing, $M = 257$).

**Lemma 2.9** If the coefficients $a = (a_0, \ldots, a_3)$ are drawn uniformly at random from \{0, \ldots, M − 1\}, then for any two different tuples $x = (x_0, \ldots, x_3)$ and $y = (y_0, \ldots, y_3)$, with $0 \leq x_i, y_j \leq M − 1$

$$Pr(h_a(x) = h_a(y)) = \frac{1}{M}.$$  

Proof: Since $x$ and $y$ are distinct, they differ in at least one position. Wlog, we can assume that $x_3 \neq y_3$.

We wish to compute the probability that $\sum_{i=0}^{3} a_i \cdot x_i \equiv \sum_{i=0}^{3} a_i \cdot y_i \mod M$ or, equivalently, that

$$\sum_{i=0}^{2} a_i \cdot (x_i - y_i) \equiv a_3(y_3 - x_3) \mod M.$$  

We are drawing $a_0, \ldots, a_3$ at random. Say, we have just drawn $a_0, a_1$ and $a_2$ and the left-hand side evaluates to $c$. The probability we are after becomes the probability that after we draw $a_3$, $a_3(y_3 - x_3) \equiv c \mod M$. Since $M$ is prime and $y_3 - x_3 \neq 0$, the probability of that is precisely $1/M$. □

In our case, the result means that if we select a random hash function from that family, the expected length of the list in the hash table entry will be just shy of one.

This result generalizes to $K$-tuples not just 4-tuples. The basic framework, now in general terms, is this.

1. We are building a list (set, database) of records that are uniquely identified by ID’s from the range $[0..N − 1]$.

2. We have some limit on the number of records we must be prepared to handle. We pick a prime number $M$ that is somewhat larger than that limit.
3. We will see integers in the range \([0..N-1]\) as integers in base \(M\), that is, as tuples of base-\(M\) digits. The number of base-\(M\) digits to represent each integer in the range \([0..N-1]\) is given by \(K = \lceil \log_M N \rceil\).

4. We will define our hash function on the set of all \(K\)-tuples \([x_0, x_1, \ldots, x_{K-1}]\) of base-\(M\) digits. That range is equivalent to the range of integers \([0..M^K-1]\) and it encompasses the range of our ID’s, \([0..N-1]\). Thus, that hash function can be used on our ID’s.

5. Specifically, to define the hash function, we select \(K\) integers \(a_0, a_1, \ldots, a_{K-1}\) independently and uniformly at random from the set \([0..M-1]\), and we define

\[
h_{a_0, \ldots, a_{K-1}}(x_0, \ldots, x_{K-1}) = \sum_{i=0}^{K-1} a_i \cdot x_i \mod M.
\]

Thus, the key property of a family \(\mathcal{H}\) of hash functions is:

For any two distinct data items \(x\) and \(y\), exactly \(|\mathcal{H}|/M\) of all hash functions in \(\mathcal{H}\) map \(x\) and \(y\) to the same bucket (entry in the array), with \(M\) being the number of buckets.

Families of hash functions with that property are universal. The discussion above demonstrated an example of a universal family. If a family of hash functions is universal, the expected length of a bucket is no more than 1! (Probability that \(x\) is hashed into a bucket \(i\) is \(1/M\) and we will have no more than \(M\) records to maintain).

3 Divide-and-conquer algorithms

This is a broad class of algorithms that follow the divide-and-conquer strategy:

1. break the problem into subproblems of the same type
2. solve these subproblems
3. appropriately combine the solutions to subproblems into a solution to the problem

Clearly, divide-and-conquer strategy results in recursive algorithms. Often, once designed, they can be rewritten into the interactive form.

The work takes place when partitioning (step 1), at the tail end of the recursion (where small instances of the problem are solved outright), and in putting the subproblem solutions together (step 3).
3.1 Multiplication

Let $x$ and $y$ be two $n$-bit binary numbers. We will assume that $n$ is a power of 2, say $n = 2^k$. If it is not so, we can pad the numbers with leading 0s (that does not change the value of the numbers nor the value of the product; it also does not affect the asymptotic estimate of the running time). Later we will show that such padding has no essential effect on the running time of the algorithm.

We can now see the two numbers $x$ and $y$ as

$$
x = \begin{array}{c|c}
x_L & x_R \\
\end{array} = 2^{n/2}x_L + x_R
$$

$$
y = \begin{array}{c|c}
y_L & y_R \\
\end{array} = 2^{n/2}y_L + y_R.
$$

Hence,

$$
xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.
$$

It follows that to compute $xy$, we need to (assuming $n > 1$; otherwise, we just return 1 if $x = y = 1$, and 0, if $x = 0$ or $y = 0$):

1. extract $x_L$, $x_R$, $y_L$ and $y_R$ from $x$ and $y$ (can be done in time $O(n)$)
2. compute $x_L y_L$ (product of two $n/2$-bit numbers)
3. compute $2^n x_L y_L$ (shift by $n$ places; can be done in time $O(n)$)
4. compute $(x_L + x_R)(y_L + y_R)$ (product of two $n/2 + 1$-bit numbers)
5. compute $x_R y_R$ (product of two $n/2 + 1$-bit numbers)
6. compute $x_L y_R + x_R y_L$ as $(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$ (two subtractions of $n/2$-bit numbers)
7. compute $xy$ by two additions of $2n$-bit numbers

The algorithm is clearly correct — it realizes the formula we have derived. If we denote by $T(n)$ the time needed by the algorithm to multiply two $n$-bit numbers, then we have the following recurrence:

$$
T(n) \leq 3T(n/2 + 1) + O(n).
$$

We also have $T(1) = O(1)$.

Is it a better algorithm then the $O(n^2)$ we saw before? YES. Next section shows why.
3.2 Recurrence relations

The recurrence relation we saw in the previous section is quite typical. We will now derive a formula solving a more general recurrence

\[ T(n) = aT(\lceil n/b \rceil) + O(n^d), \]

where \( a \) is a positive integer, \( b > 1 \) and \( d > 0 \). It turns out that the solution we obtain is also a solution to recurrences given by the inequality \( \leq \) and depending on \( \lceil n/b \rceil + c \), where \( c \) is a positive integer.

**Theorem 3.1 (Master theorem)***

If \( T(n) = aT(\lceil n/b \rceil + \epsilon) + O(n^d) \), where \( a \) is a positive integer, and \( b > 1 \), \( d > 0 \) and \( \epsilon \) are reals, then

\[ T(n) = \begin{cases} 
O(n^d) & \text{if } d > \log_b a \\
O(n^d \log n) & \text{if } d = \log_b a \\
O(n^{\log_b a}) & \text{if } d < \log_b a 
\end{cases} \]

If \( O \) is replaced by \( \Theta(\Omega) \) in the recurrence, \( O \) can be replaced with \( \Theta(\Omega) \) in the assertion.

**Proof:** In the proof, we will assume that \( n = b^k \). That does not influence the final result but makes computations simpler.

We will visualize the algorithm in terms of the recursion tree. In the root, we have \( T(n) \). It consists of \( O(n^d) \) (the contribution of the level 0) and of the contributions from lower levels, the sum of \( a \) terms \( T(n/b) \). Each of these values, can be viewed in the same way. They are the sum of \( O((n/b)^d) \) (the 1st-level contribution) and (again) \( a \) terms of (this time) \( T(n/b^2) \). Thus, if we attribute to each internal node in the tree at level \( m \) the quantity \( O((n/b^m)^d) \), and \( T(1) \) to all leaves (level \( k \)), then \( T(n) \) is the sum of the quantities allocated to all nodes.

This sum can be computed level by level. There are \( k + 1 \) levels 0, \ldots, \( k \). The sum for the level \( m < k \) is

\[ a^m O((n/b^m)^d) = O(n^d) \cdot (a/b^d)^m \]

The sum for the level \( k \) is

\[ a^k T(1) = a^{\log_b n} T(1) = n^{\log_b a} T(1) = O(n^{\log_b a}). \]

It follows that

\[ T(n) = O(n^{\log_b a}) + \sum_{m=0}^{k-1} O(n^d) \cdot (a/b^d)^m = O(n^{\log_b a}) + O(n^d) \sum_{m=0}^{k-1} (a/b^d)^m. \]

Thus, If \( a = b^d \) or, equivalently, \( d = \log_b a \), then

\[ T(n) = O(n^{\log_b a}) + k \cdot O(n^d) = O(n^d + n^d \cdot \log_b n) = O(n^d \log n). \]
If \( a < b^d (\log_b a < d) \), then
\[
T(n) \leq O(n^{\log_b a}) + O(n^d)C
\]
for some constant \( C \).

If \( a > b^d (\log_b a > d) \), then
\[
T(n) \leq O(n^{\log_b a}) + O((a/b^d)^k) = O(n^{\log_b a}) + O(n^{\log_b a}/n^d) = O(n^{\log_b a}).
\]
\[\square\]

Since \( \log_2 3 > 1 \), the running time of the multiplication algorithm from the previous section is \( O(n^{\log_2 3}) = O(n^{1.59}) \) — much better than \( O(n^2) \).

### 3.2 Binary search

This can be viewed as a restricted dictionary problem, where we are only concerned with accesses (no inserts and deletes).

Given a sorted array \( A[0..n-1] \) of keys and a key \( k \), find if \( k \) is in the array and, if so, find its position. The recurrence is:

\[
T(n) = T(\lceil n/2 \rceil) + O(1)
\]

It solves to \( T(n) = O(\log n) \).

### 3.3 Sorting

Most sorting algorithms can be represented as divide-and-conquer algorithms. We will discuss here the insertion, selection, mergesort and quicksort sorting algorithms. In what follows, we assume that \( a[1..n] \) is an array of \( n \) integers (but could also be an array of reals or strings).

#### 3.3.1 Insertion sort

The algorithm is based on the idea to sort the subarray \( a[1..n-1] \) and insert the last item, \( a[n] \), in the correct place (hence, insertion sort).

```plaintext
function insertionSort(a[1..n])
Input: An array of numbers a[1..n]
Output: a sorted version of this array

for i = 2 to n:
  k = i - 1
  x = a[i]
  while x < a[k]:
```
\begin{align*}
  a[k + 1] &= a[k] \\
  k &= k - 1 \\
  a[k + 1] &= x
\end{align*}

return \ a

Denoting by $T(n)$ the number of comparisons required by insertion sort, we get that $T(2) = 1$ and $T(n) \leq T(n - 1) + (n - 1)$. Thus, $T(n) \leq n(n - 1)/2$. Moreover, the running time is given by $O(n^2)$.

The bound for the number of comparisons needed can be attained (for the sequence in reversed sorted order). The best case performance is much better. It takes only $n - 1$ comparisons to sort a sequence that is already sorted (and $O(n)$ time).

On a random arrangement of distinct integers, average number of comparisons is asymptotically equal to $n^2/4$.

Here “partitioning” the problem into subproblems is trivial, “merging” is costly. Note the imbalance in the sizes of the two problems.

### 3.3.2 Selection sort

The idea here is to select in an unsorted sequence the smallest element, place it in its right place and complete the task by sorting the remaining elements.

```python
function selectionSort(a[1..n])
Input: An array of numbers a[1..n]
Output: a sorted version of this array

for i = 1 to n - 1:
  k = findMin(a[i..n])
  swap elements in positions i and k
return a
```

The procedure `findMin` finds the first integer $k$ such that $a[k]$ is minimum in the input array. Clearly, it requires $n - i$ comparisons and runs in $O(n)$ steps. Thus the total number of comparisons for the selection sort is $(n - 1)n/2$, and the best and average numbers are the same.

Here “partitioning” is costly and “merging” is easy. Again, note the imbalance in the sizes of two problems.

### 3.3.3 Mergesort

This is a sorting algorithm that partitions the numbers to be sorted into two (almost) equal parts, sorts the parts, and merges the sorted parts into a single sorted list. Can be viewed as a balanced version of the insertion sort.
function mergesort(a[1..n])
Input: An array of numbers a[1..n]
Output: a sorted version of this array
if n > 1:
    return merge(mergesort(a[1..⌊n/2⌋]), mergesort(a[⌊n/2⌋+1..n]))
else:
    return a

The correctness is evident, as long as merge works correctly. Thus, let’s show how to implement merge:

function merge(x[1..k], y[1..l])
Input: Two sorted arrays x[1..k] and y[1..l]
Output: a sorted array obtained from merging x and y
if k = 0: return y
if l = 0: return x
if x[1] ≤ y[1]:
    return x[1] ◦ merge(x[2..k], y[1..l])
else:
    return y[1] ◦ merge(x[1..k], y[2..l])

The correctness is clear. The time per recursive call is constant (assuming that the required array space is allocated in advance). Thus, the running time is $O(k + l)$.

It follows that the running time of mergesort, $T(n)$, satisfies

$$T(n) \leq 2T(\lceil n/2 \rceil) + O(n)$$

and so, $T(n) = O(n \log n)$.

Once we look at how mergesort works, an iterative version emerges. It uses the data structure queue (of linked lists), with two basic operations: enqueue and dequeue.

function it-mergesort(a[1..n])
Input: An array of numbers a[1..n]
Output: a sorted version of this array
Q = []
for i = 1 to n:
    enqueue(Q, [a[1]])
while |Q| > 1:
    enqueue(Q, merge(dequeue(Q), dequeue(Q)))
return dequeue(Q)
3.3.4 Quicksort

This is a very fast sorting algorithm also based on the divide-and-conquer approach. It is broadly used (in particular, in unix sort command. In some sense, can be viewed as the correct (“balanced” partition) implementation of the selection sort.

```
function quicksort(a[1..n])
Input: An array of numbers a[1..n]
Output: a sorted version of this array
if n ≤ 50:
    return insertionsort(a[1..n])
pivot = a[i], where i is a random integer from [1..n]
partition(a[1..n], pivot, j, k)
return quicksort(a[1..j − 1]) ◦ a[j..k] ◦ quicksort(a[k + 1, n])
```

Here partition(a[1..n], pivot, j, k) rearranges elements in a[1..n] so that all elements < pivot are in a[1..j − 1], all elements = pivot are in a[j..k], and all elements > pivot are in a[k + 1..n]. One implementation collects elements that are < pivot, = pivot and > pivot into three linked lists and then puts the lists together. This algorithm results in a stable version of quicksort.

There are linear time implementations of partition that are in-place but they are not stable!!

In the worst case, quicksort is a quadratic algorithm. But the expected running time is $O(n \log n)$.

Here is why. Let $T(n)$ be the expected running time of quicksort when sorting $n$ elements. Then, we have the following recurrence:

$$T(n) = \Theta(n) + \frac{1}{n} T(n − 1) + \frac{1}{n} \sum_{k=1}^{n-2} (T(k) + T(n − 1 − k)) + \frac{1}{n} T(n − 1)$$

$$= \Theta(n) + \frac{2}{n} \sum_{k=1}^{n-1} T(k)$$

We will prove by induction that for some positive $a$ and $b$, $T(n) \leq an \log n + b$. The basis is evident if $b \geq T(1)$. Before we start the inductive step, we note that

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{[n/2]-1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k.$$

Since $\lceil n/2 \rceil − 1 \leq n/2$, the first term in the sum is bounded by

$$\sum_{k=1}^{[n/2]-1} k \log k \leq \log(n/2) \sum_{k=1}^{[n/2]-1} k.$$
and the second on by
\[ \sum_{k=1}^{n-1} k \log k \leq \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k. \]

Thus,
\[ \sum_{k=1}^{n-1} k \log k \leq \log n \sum_{k=1}^{\lceil n/2 \rceil-1} k \leq \frac{1}{2} n^2 \log n - \frac{1}{2} ([n/2] - 1)[n/2]. \]

Since \((\lceil n/2 \rceil - 1)[n/2] \geq (n/2 - 1)(n/2),\)
\[ \sum_{k=1}^{n-1} k \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{4} n^2 + \frac{1}{4} n. \]

It follows that
\[
T(n) = \Theta(n) + \frac{2}{n} \sum_{k=1}^{n-1} T(k) \leq \Theta(n) + \frac{2}{n} \sum_{k=1}^{n-1} (ak \log k + b)
\leq \Theta(n) + \frac{2a}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 + \frac{1}{4} n \right) + 2b
= an \log n + (2b - \frac{an}{4} + \Theta(n)).
\]

Thus, if \(a\) is selected to be large enough,
\[ T(n) \leq an \log n + b, \]
completing the inductive argument.

Note that this version of quicksort is an example of a randomized algorithm (robust to any possible structure of the input sequence). Note also, that instead of always choosing a random element from the array as the pivot, we could simply randomly rearrange the input sequence in the preprocessing phase. After that we could always use a predetermined element as the pivot (the first in the array, the last in the array, the middle one, even the average of the three).

### 3.3.5 Sorting by decision trees

This is a broad class of algorithms that are described by binary trees. Given a sequence \(a_1, \ldots, a_n\), we descend down the tree, choosing our path based on the values \(a_i\). Eventually, we reach a leaf and we return the ordering associated with that leaf. Each internal node in the tree is a decision point of the form “is \(a_i < a_j\)?” If the answer is No, we descend to the left child. Otherwise, we descend to the right child. Here is an example for \(n = 3\). Any input \(a_1, a_2, a_3\) takes us down to a leaf, which specifies a correct (sorted) ordering.
Algorithms that can be so represented are called comparison sort algorithms. Many (but not all) sorting algorithms are of that type (for instance, mergesort, insertionsort, selectionsort, quicksort).

For algorithms of that type we can get a lower bound on time they need to sort (in the worst case). If a comparison sort for \( n \) numbers is correct, then its tree must have at least \( n! \) leaves. The algorithm makes in the worst case \( \Theta(p) \) comparisons, where \( p \) is the maximum number of internal nodes on a path from the root to a leaf in the tree. Thus, the worst-case running time is \( \Omega(p) \).

The tree is binary, so it has at most \( 2^p \) leaves. Thus, \( n! \leq 2^p \) and so,

\[
\log n! \leq p.
\]

Since \( n! \geq (n/2)^{n/2} \), there is a constant \( c > 0 \) such that

\[
c \cdot n \log n \leq p
\]

It follows that the worst-case running time of any comparison sort is \( \Omega(n \log n) \).

### 3.4 Counting sort, radix sort, stable sorting algorithms

Counting sort applies to sequences \( x_1, \ldots, x_n \) of integers from a “small” range, say \( \{0, 1, \ldots, k-1\} \). To sort such an element, we initialize an array \( C[0..k-1] \) of counters by setting each \( C[i] \) to 0. Then, we traverse the sequence. For each element \( x_i \) we increment by 1 the number of elements equal to \( x_i \) in the sequence: \( C[x_i] = C[x_i] + 1 \). Once we are done, we form a new sequence with \( C[0] \) 0s, \( C[1] \) 1s, etc. The algorithm works in time \( \Theta(n + k) \).

Radix sort is designed to sort \( n \) \( m \)-tuples of elements from \( [0..k-1] \) lexicographically (leftmost positions are more significant than rightmost ones). The algorithm consists of \( m \) rounds of counting sort with respect to elements in positions starting at the very right and proceeding towards left. The algorithm works in time \( \Theta(m(n + k)) \).

A sorting algorithm is stable if it does not change the relative order of elements with the same key.
3.5 Selection problem, computing the median

We will now deal with another problem related to relative values of elements in a set — the selection problem: given an array \( a[1..n] \) of numbers, and an integer \( k \) such that \( 1 \leq k \leq n \), return the \( k \)-th smallest element in the array.

If \( k = 1 \) or \( k = n \), then it can be done in time \( O(n) \). But we are often interested in the median, which is defined as the \([n/2]\)-smallest element. Can we find the median in linear time?

Here is a divide-and-conquer approach to the general selection problem.

\[
\text{function selection}(a[1..n], k)
\]
\[\text{Input: An array of numbers } a[1..n], \text{ an integer } 1 \leq k \leq n\]
\[\text{Output: the } k \text{-th-smallest element in } a[1..n]\]
\[\text{pivot} = a_i, \text{ where } i \text{ is a random integer from } [1..n]\]
\[\text{partition}(a[1..n], \text{pivot}, j, m)\]
\[\text{if } k < j: \text{ return selection}(a[1..j−1], k)\]
\[\text{if } k > m: \text{ return selection}(a[m+1..n], k−m)\]
\[\text{else: return } a[j]\]

In the worst case, the algorithm may perform as many as \( \Theta(n^2) \) steps (when we want to choose the smallest element, all elements are distinct and in each step the largest element is chosen as the pivot).

We will now analyze the expected time of the algorithm, say \( T(n) \). A pivot is good if it lies within the 25th to 75th percentile of the array it is chosen from (formally if it is at least as large as the \( \lceil 0.25n \rceil \)-th smallest element, and at most as large as the \( \lfloor 0.75n \rfloor \)-th smallest element).

The probability that a randomly selected pivot is good is at least 0.5.

**Lemma 3.2** On average a fair coin needs to be tossed two times for a “heads” to show up.

Proof: Let \( E \) be the expected number of tosses for “heads” to be seen. We definitely need at least one toss. If it is “heads” we are done; otherwise, (with the probability 1/2), we still expect \( E \) tosses. Thus, \( E = 1 + \frac{1}{2}E \), which works out to \( E = 2 \).

Therefore, after two calls to partition, the array will shrink on average to no more than 75% of its range.

Now, we have the following statement: time taken on array of size \( n \leq \) time taken on an array of size \( 3n/4 \) + time to reduce the array size to \( \leq 3n/4 \).

Taking expected values of both sides (and using the lemma for the second term on the RHS), we get

\[
T(n) \leq T(3n/4) + O(n).
\]

That implies \( T(n) = O(n) \).

The median is a very useful statistic of a set, in many respects better than the average. It is always one of the elements in the set, and it is robust to outliers.

A slightly more elaborate analysis applies to quicksort.


4 Rooted Trees

A rooted tree is given by its set of vertices, each of which has a certain number (possibly 0) of children. Moreover, we require that that no node is its own proper descendant (no cycles along “child” pointers) and no node has more than one immediate ancestor; and that every two nodes have a common ancestor. The latter implies that there is a node in the tree, called the root, with all other nodes being its descendants. Nodes that have no children are called leaves.

We represent arbitrary rooted trees by a linked data structure. Each node contains satellite data, and pointers left (leftmost child), next (next sibling), and p (parent). Rooted trees can also be represented as graphs (we will discuss that matter later).

A particularly important class of rooted trees is the class of binary trees. In a binary tree, each internal node has up to two children, which we call the left and the right child, respectively.

We represent binary search trees by a linked data structure. Each node contains satellite data, and fields left (left child), right (right child), and p (parent).

We note that nodes of a binary tree can be visited in an inorder, postorder and preorder way. The inorder walk is accomplished by the following algorithm:

```
function inorderWalk(x)
Input: an object x (of type of the node of a binary tree)
Output: depends on actions performed during the walk
if x ≠ nil:
    inorderWalk(left[x])
    print key[x]
    inorderWalk(right[x])
return
```

The algorithm runs in time Θ(n) (proof by induction). Postorder and preorder walks differ only by the placement of the print command.

The preorder and postorder walks generalize to the case of arbitrary rooted trees.

4.1 Operations on binary search trees

A binary tree is a binary search tree if its satellite data contains a distinguished field key (storing an integer, real, or string) and if the following condition holds: for every node x, if y is a node in the left subtree of x, key[y] ≤ key[x], and for every node z in the right subtree of x, key[x] ≤ key[z].

The first operation searches for an object with the key k.

```
function treeSearch(x,k)
Input: an object x (of type of the node of a binary tree)
k a value from the data type used for keys
```
Output: the object whose key is $k$, or nil, if no object has key $k$
if $x = \text{nil}$ or $k = \text{key}[x]$: return $x$
if $k < \text{key}[x]$: return treeSearch(left[x], k)
else: return treeSearch(right[x], k)

An iterative code looks as follows:

```
function itTreeSearch(x, k)
Input: an object $x$ (of type of the node of a binary tree)
   $k$ a value from the data type used for keys
Output: the object whose key is $k$, or nil, if no object has key $k$
while $x \neq \text{nil}$ and $k \neq \text{key}[x]$:  
   if $k < \text{key}[x]$: $x = \text{left}[x]$
   else: $x = \text{right}[x]$
return $x$
```

The running time is given by $\Theta(m)$, where $m$ is the length of the search path. Thus, it is also bounded by $O(h)$, where $h$ is the height of the tree.

Other useful operations are: treeMax(x), treeMin(x), treeSucc(x) and treePrec(x).

Finally, we have procedures for inserting elements into a binary search tree and for deleting them.

```
function treeInsert(T, z)
Input: a binary tree $T$ and an object $z$ to insert
Output: upon termination, $T$ represents a binary tree with $z$ inserted
$y = \text{nil}$
$x = \text{root}[T]$
while $x \neq \text{nil}$:
   $y = x$
   if key[z] < key[x]:
      $x = \text{left}[x]$
   else:
      $x = \text{right}[x]$
$p[z] = y$
if $y = \text{nil}$:
   if $\text{root}[T] = \text{nil}$:  
      $\text{root}[T] = z$
   else:
      if key[z] < key[y]
```

$y$ is to be a parent of $x$ as we go down the tree and look for the place to insert $z$. When done, we will be attaching $z$ to $y$.

When the loop ends, $x = \text{nil}$. $y$ - the parent of $x$, possibly nil.

$T$ was empty.
Tree delete works as follows. It assumes we already found the node, say \( z \), to delete. If that node has at most one child, the deletion is easy. If that node has two children, we find the immediate successor of \( z \), say \( y \). It swaps the content of \( z \) and \( y \) and removes \( y \).

function tree-Delete\((T, z)\)
Input: a binary tree \( T \) and an object \( z \) to delete
assumed to be in the tree
Output: upon termination, \( T \) represents a binary tree
with \( z \) deleted
if \( \text{left}[z] = \text{nil} \) or \( \text{right}[z] = \text{nil} \):
    \( y = z \)
else
    \( y = \text{treeSucc}(z) \)
if \( \text{left}[y] \neq \text{nil} \):
    \( x = \text{left}[y] \)
else:
    \( x = \text{right}[y] \)
if \( x \neq \text{nil} \):
    \( p[x] = p[y] \)
if \( p[y] = \text{nil} \):
    then root\([T]\) = \( x \)
else:
    if \( y = \text{left}[p[y]] \):
        \( \text{left}[p[y]] = x \)
    else:
        \( \text{right}[p[y]] = x \)
if \( y \neq z \):
    \( \text{key}[z] = \text{key}[y] \)
% If \( y \) has other fields, copy them, too

4.2 Balanced binary search trees

All operations we described for binary trees (except for those that are to traverse them) run in time \( O(d(T)) \) where \( d(T) \) is the depth of the tree, that is, the length of a longest path from the root to a leaf.
If we build a binary search tree in a sequence of insertions, if the new nodes come in the ascending order (or descending, or something close to these two sorted orders) the tree will become “imbalanced” and will assume the shape not far from a linked list.

When that happens, the best we can say about the running time of the key binary search tree operations is that they take $O(n)$ time, where $n$ is the number of elements in the tree. This is quite bad when compared with the performance of these operations when trees are balanced.

What to do? Modify the insert and delete operations so that whenever they introduce an imbalance, they spend some extra time fixing the problem - bringing the tree back to the balanced state. This is typically accomplished by a sequence of rotation operations. The first versions of insert and delete operations that maintain the tree in a balanced state were proposed by G. M. Adelson-Velskii and E. M. Landis (Soviet scientists) who described it in 1962. The trees one sees when using their schema are quite strongly balanced (as strongly as one might hope for) with each node having the absolute value of the difference between the depths of its left and right subtrees bounded by 1. A similar structure of red-black trees was invented by Rudolf Bauer in 1972 (the name red-black trees comes from a paper by Guibas and Sedgewick from 1978). The formal definition is as follows. In a red-black tree (in which we assume that leaves are empty nodes pointed to by nil pointers of their parents):

1. Each node is either red or black, with the root and all leaves being black
2. Both children of every red node are black
3. Every path from a root node to any of its descendant leaves contains the same number of black nodes (that implies the property for every node in the tree)

The key property of balanced trees is that if they store $n$ nodes, their height is $O(\log n)$. We will show that property for AVL trees. Let us denote by $T(h)$ the minimum number of nodes in an AVL tree of height $h$. Then, $T(0) = 1$, $T(1) = 2$ and

$$T(h) = T(h-1) + T(h-2) + 1 \quad \text{for } h \geq 2.$$  

Setting $S(h) = T(h) + 1$, we get $S(0) = 2$, $S(1) = 3$ and

$$S(h) = S(h-1) + S(h-2) \quad \text{for } h \geq 2.$$  

Thus, $S(h) = F_{h+3} \geq 2^{0.69(h+3)}$. It follows that if an AVL tree of height $h$ has $n$ nodes then

$$n \geq S(h) \geq 2^{0.694(h+3)}.$$  

Consequently,

$$\log n \geq 0.694(h + 3)$$

and

$$h \leq 1.45 \log n - 3.$$
The essence of the rotation operation on binary search trees is captured by the diagram in Figure 1.

When applied to any node of a binary search tree, a rotation results in a binary search tree.

After we insert a node into a tree, it can get out of balance at some node up the path to the root. To rebalance the tree, we move up that path, updating the balance indicators (integers $-1$, $0$ and $1$; $-1$ indicates the left subtree is higher, etc.). If at some point we find a node which is not balanced rotations bring it back into balance and the process stops. The essence of the method is given in Figure 2.

The operation of deletion with rebalancing is more complex to describe. For instance, it does not necessarily end the first time rotations are used but may continue all the way up to the root. But it is really quite similar.
5 Representing graphs

By explicit lists of nodes and edges. The first list is needed in case of isolated vertices.
  Adjacency arrays.
  Adjacency lists.
  Show how these structures support basic graph tasks.

6 Exploring graphs — depth-first search

Why graphs? They offer a way to extract and represent most essential aspects of a problem. For instance, if we are to schedule final exams, all that matters is what exams are to be scheduled and which pairs of exams cannot be scheduled in the same time. This information can be represented by a graph. In that graph, vertices represent exams, and edges (lines connecting vertices) represent conflicts.

Formally, a graph is determined by its set of vertices (also called nodes) $V$, and by the set of edges $E$, where and edge is an unordered or ordered pair of vertices (not necessarily distinct). Edges connect its vertices (or run between them). If the two vertices in a pair are identical, the edge is called a loop. We will normally consider loopless graphs.

If all edges in a graph are unordered pairs, the graph is undirected; if all edges in a graph are ordered pairs, the graph is directed. If the graph is undirected, we identify $(u, v)$ and $(v, u)$. That is, neither $u$ nor $v$ is in any way distinguished. If the graph is directed, there is a difference between $u$ and $v$ in $(u, v)$. In general (unless $(u, v)$ is a loop), $(u, v)$ and $(v, u)$ are different. We call $u$ the tail of $(u, v)$, and $v$ the head of $(u, v)$.

6.1 How to represent graphs?

A graph can be given by the lists of its vertices and edges. This representation closely reflects the definition. But it not used often as it does not support efficient answers to some basic question concerning the structure of the graph. For instance, if you want to know if there is an edge between two vertices $u$ and $v$, you may have to traverse the entire list of edges to determine the answer.

If queries of that sort are essential to an algorithm for a problem on graphs, we may represent a graph by an adjacency matrix (or, adjacency array). We assume that we are given a fixed enumeration of vertices of the graph as $v_1, \ldots, v_n$, and define the $(i, j)$th entry in the adjacency matrix, $1 \leq i, j \leq n$, as

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

If $G$ is undirected, the adjacency matrix is symmetric.

The queries concerning the existence of an edge between a given pair of vertices can now be answered in time $O(1)$. On the other hand, the representation requires $O(n^2)$ space.
Is it good enough or is it too much? If the graph has $10^{10}$ nodes (as the World Wide Web graph at present does), it is way too much and the method is useless. We would need to allocate for it a boolean array with $10^{20}$ entries, which would require millions of terabytes of memory — unrealistic now and for quite some time to come.

But the World Wide Web contains only about $3 \cdot 10^{10}$ hyperlinks (edges). It is very very sparse. So, our first straightforward representation might actually work. Assuming about 4 bytes per node and 8 bytes per edge, it would only take about $4 \cdot 10^{10} + 24 \cdot 10^{10}$ bytes or about $28 \cdot 10^{10} \approx 280 GB$ of space. That is not out of the question.

But, our first representation (it requires $\Theta(n + m)$ memory locations, assuming each vertex index can be represented by a small number of bytes, for instance, 4) has, as noted, several drawbacks. Here is a better representation: *adjacency lists* representation. It consists of $|V|$ adjacency lists, one per each vertex. The list for a vertex $u \in V$ consists of all vertices $v$ that can be reached from $u$ by an edge.

In a directed graph, each edge $(u, v)$ is represented on exactly one adjacency list (that of the vertex $u$). In an undirected graph, each edge $(u, v)$ is represented twice.

It follows that this representation requires $O(n + m)$ memory locations (as the first one). But now, we have a quick access to all edges starting in a vertex, say $u$. We can enumerate them in time $O(d_u)$, where $d_u$ is the length of the adjacency list for $u$. In the same time, we can also determine whether there is an edge between $u$ and $v$ (but it is not $O(1)$ as for adjacency matrix).

### 6.2 Depth-first search

The problem is to visit all vertices in the graph following its edges. There are two basic approaches: depth-first search and breadth-first search. In this section, we focus on the former one.

The motivation comes from exploring mazes. At any time, we can only see what is directly around us. Thus, when we are at a fork, we know about all alternative ways in which we can continue exploration. In order to explore a maze, we need a ball of string and a piece of chalk. As we discover “intersections”, we mark them with chalk. When we are at an intersection whose all connections we already followed (we can discover that by seeing chalk at their ends), we have to go back to the place, from which we came. Rewinding the string helps in that. Otherwise, we select an unexplored connection and follow it. In this way, we will explore the part of the maze that can be reached from where we started.

That idea can be formulated as a way to explore graphs. All we need is a way to mark nodes of a graph as visited (a boolean array will do) and a stack that will play the role of the string; pushing corresponds to unwinding the string, popping to rewinding (instead of using the stack explicitly, we can and will use recursion).

```plaintext
procedure explore(v)
% Assumes a global variable G = (V,E), a graph to explore
% and a global boolean array visited
```
Input: a vertex \( v \) (an element of a graph \( G = (V,E) \)) such that \( \text{visited}(v) \) is false
Output: \( \text{visited}(u) \) is set to true for all vertices reachable from \( v \) by paths not traversing vertices marked visited

\[
\text{visited}(v) = \text{true} \\
\text{previsit}(v) \\
\text{for each edge } (v,u) \in E: \\
\text{if not visited}(u): \text{explore}(u) \\
\text{postvisit}(v)
\]

For now, we assume that \text{previsit}(v) and \text{postvisit}(v) do nothing (they are optional). Later we will exploit them in some quite clever ways.

Is the method correct? We claim that \text{explore}(v) sets \text{visited}(u) to true if and only if there is a path in \( G \) from \( v \) to \( u \) not traversing any vertices marked visited.

Since we can only proceed forward along an edge starting in a vertex that the procedure marked visited, if \text{visited}(u) was set to \text{true} by the procedure, there is a path from \( v \) to \( u \) whose all vertices are not visited before the procedure is called.

To prove the converse statement, we proceed by induction.
Method 1. Namely, we prove by induction on \( k \), that for every graph \( G \) with \( k \geq 1 \) non-visited vertices, the call to \text{explore}(v), where \( v \) is not visited, marks as visited all vertices reachable from \( v \) by paths traversing only vertices that are not visited at the time of the call.

For \( k = 1 \) the claim is obvious. The only vertex reachable from \( v \) by such paths is \( v \) itself (the only unmarked vertex in \( G \)). Thus, let us assume \( k > 1 \) and let \( u \) be a vertex reachable from \( v \) by a “non-visited” path. Let \( z \) be the first non-visited neighbor of \( v \) from which \( u \) is reachable by a “non-visited” path (according to the order in which the neighbors of \( v \) are considered). No vertex of this path is marked visited by an earlier call to \text{explore} (otherwise \( z \) is not the first neighbor of \( v \) from which \( u \) is reachable). By induction, the call \text{explore}(z) marks \( u \) visited.

To prove the converse statement, we proceed by induction. Namely, we prove that for every \( k \geq 0 \), for every vertex \( u \), if there is a path from \( v \) to \( u \) of length at most \( k \) and not going through vertices marked visited, then \text{visited}(u) will be set to \text{true}.

For \( k = 0 \) there is only one such vertex \( u \), namely \( u = v \) and \text{visited}(v) is indeed set to \text{true}. Let us assume that the claim holds for some \( k \geq 0 \) and let us consider a vertex \( u \) such that there is a path from \( v \) to \( u \) of length at most \( k + 1 \) and not going through vertices marked visited. If the length of that path is at most \( k \), we are done by the induction hypothesis. If the length of that path is \( k + 1 \), let \( z \) be the predecessor of \( u \) on that path. By the induction hypothesis, \text{visited}(z) was set to \text{true} by the procedure \text{explore}(v). It must have been done in the recursive call \text{explore}(z). Clearly, when the loop is iterated for the edge \((z,u)\), either \text{visited}(u) = \text{true}, which means it has already been set to
true earlier in the execution of explore(v), or visited(u) = false and it will be set to true now.

We will now use the procedure explore in the depth-first search algorithm.

```
procedure dfs(G)
Input: a graph G = (V, E))
Output: visited(u) is set to true for every u ∈ V
Other information computed by pre- and postvisit

for all v ∈ V:
    visited(v) = false

for all v ∈ V:
    if not visited(v): explore(v)
```

Ties are broken alphabetically (or according to ≤ ordering). The correctness is clear. To estimate the running time, we make the following observations.

1. The first loop takes \(O(n)\) steps (as always when we discuss graphs, \(|V| = n\) and \(|E| = m\)).
2. In the second loop, each vertex is visited exactly once (thanks to the visited array).
3. During the exploration of a vertex, some fixed amount of work is done by pre/postvisit,
4. In addition, adjacent edges are scanned to see if they lead to a new place.
5. The loop takes different time for different vertices. But looking at all vertices together, we see that all these loops have \(O(m)\) iterations (\(2m\) for undirected graphs, \(m\) for directed graphs).

It follows that the running time is \(O(n + m)\).

### 6.3 Connected components in undirected graphs

The dfs procedure is easy to adapt for the task of assigning to each vertex \(u\) the number of its connected component, \(ccnum[u]\). In the main procedure, we need to set \(ccnum\) to 0, and set up \(cc\) to 0, too (\(cc\) is the counter of connected components). Each time the second loop discovers a new vertex, \(cc\) is incremented by 1. Finally, we define previsit as follows.

```
procedure previsit(v)
ccnum[v] = cc
```

When the procedure terminates, \(cc\) contains the number of connected components. Moreover, \(ccnum[u]\) is the number of the connected component containing \(u\).
6.4 Previst and postvisit orderings

We will now code previsit and postvisit so that they compute the time of discovery of a node and the time of the final departure. To this end, we introduce a counter clock, initially set to 1.

```plaintext
procedure previsit(v)
pre[v] = clock
clock = clock +1

procedure postvisit(v)
post[v] = clock
clock = clock +1
```

The figure below shows the effect these procedures have.

![Graph with intervals](image)

We have the following property.

**Proposition 6.1** For any nodes u and v, the two intervals \([pre(u), post(u)]\) and \([pre(v), post(v)]\) are either disjoint or one is contained within the other.

Why? Because \([pre(u), post(u)]\) is essentially the time during which vertex u was on the stack. The last-in, first-out behavior of a stack explains the rest.

Finally, we note that once we run dfs on a graph, all edges are either in the forest of trees, or are back edges (they go back to a vertex already discovered).

1. If \([pre(u), post(u)]\) and \([pre(v), post(v)]\) are disjoint, u and v are not in the ancestor/descendant relation wrt to the dfs forest

2. Otherwise, one interval is included in the other (by the property above). If, for instance, \([pre(u), post(u)] \subseteq [pre(v), post(v)]\), then u is the descendant of v in the dfs tree, to which both nodes belong
6.5 Dfs in directed graphs

There is no change in the formulation of the algorithm. The figure below illustrates the process.

We now have four types of edges: edges that are in the dfs forest; forward edges from a node to a non-child descendant in the forest; back edges from a node to its ancestor in the dfs forest; and cross edges from a node to a node completely explored (unrelated to the node by the ancestor/predecessor relation).

Here is how the type of an edge \((u, v)\) is determined by \text{pre/postvisit} times.

\[
\begin{array}{ccc|c}
\text{u} & \text{v} & \text{v} & \text{u} \\
\hline
\text{Tree/forward} & [ & [ & ] \\
\text{Back} & [ & [ & ] \\
\text{Cross} & [ & [ & ] \\
\end{array}
\]

6.6 Directed acyclic graphs (dag’s)

A \textit{cycle} in a directed graph is a sequence of vertices \(v_0, \ldots, v_{n-1}\) such that \((v_0, v_1), \ldots, (v_{n-2}, v_{n-1})\) and \((v_{n-1}, v_0)\) are edges in the graph. A directed graph with no cycles is \textit{acyclic}. The common abbreviation is: a \textit{dag}.

\textbf{Property.} A directed graph has a cycle if and only if its depth-first search reveals a back edge.

\textbf{Proof:} If depth-first search reveals a back edge, then a cycle exists (by the definition of a back edge). Conversely, let \(v_0, \ldots, v_{n-1}\) be a cycle. Let \(v_i\) be the first node from the cycle discovered by the dfs (with the lowest previsit time). All other nodes on the cycle are reachable from \(v_i\) and so, are it descendants in the tree. But then, the edge \((v_{i-1}, v_i)\) (or \((v_{n-1}, v_0)\), if \(i = 0\)) is a back edge.

\textbf{Property.} In a dag, every edge leads to a node with a lower \text{post} number.

\textbf{Proof:} In a dag there are no back edges. But tree, forward and cross edges have precisely the property in question.
Thus, dags can be linearized or topologically sorted. That is, the nodes can be labeled so that each edge leads to a node with a lower label (enough to label the nodes 1, \ldots, n according to decreasing post numbers. It also holds that only dags can be linearized.

Clearly, the last node in that labeling is a sink, that is, has no outgoing edges; while the first node is a source, has no incoming edges. A dag can have more than one source and sink. One can use that property as an alternative algorithm for linearization (testing acyclicity).

### 6.7 Strongly connected components

Two nodes in a directed graph are connected if there is a path from $u$ to $v$ and a path from $v$ to $u$.

The relation of being connected is an equivalence relation. Thus, it partitions the set of vertices of the graph into maximal sets of mutually connected vertices. Such partition is unique. Its elements are called strongly connected components. Moreover, a directed graph and its strongly connected components determine the dag of strongly connected components.

![Diagram of a directed graph](image)

**The problem.** Compute strongly connected components.

Here are some key properties we will use to design an efficient algorithm.

**Property 1.** If the `explore` subroutine is started at node $u$, then it will terminate precisely when all nodes reachable from $u$ have been visited.

Thus, if we start `explore` in a node from the sink scc, all vertices in that scc will be visited (identified) and only those.

Thus, two questions: (A) how to find a node we are certain belongs to a sink scc and (B) how do we continue once the first has been discovered?

There is no way to quickly identify a node in a sink scc. But we can easily find a node in a source scc.

**Property 2.** The node that receives the highest post number in a dfs lies in a source scc.

This is a corollary from a more general property.
Property 3. If \( C \) and \( C' \) are scc’s and there is an edge from a node in \( C \) to a node in \( C' \), then the highest post number in \( C \) is bigger than the highest post number in \( C' \).

Proof: If the first node from \( C \cup C' \) visited by a dfs, say \( x \), is in \( C \), then the procedure \texttt{explore} gets stuck only after all of \( C \) and \( C' \) has been visited. Therefore, \( x \) will have a higher post value than any node in \( C' \), and the assertion follows.

Otherwise, \( x \in C' \). In that case, \texttt{explore} gets stuck after seeing all of \( C' \) (and setting the post values for all the nodes of \( C' \)) before visiting an node in \( C \). Thus, the post number for any node in \( C \) is higher than those for nodes in \( C' \).

Thus, to find a node in a sink scc, we need to produce \( G^R \) (a graph obtained from \( G \) by reversing the direction of each edge). That graph has the same scc’s as \( G \) but source and sink components are swapped.

On to the problem (B). Once the first component has been found and removed, the highest post number (wrt \( G^R \)) among the remaining nodes identifies a node in the sink of \( G \) minus that first component. So, starting explore there, gives us the next component, etc.

So, the algorithm looks as follows:

1. Run dfs on \( G^R \) to compute post numbers
2. Run the undirected connected component algorithm on \( G \), and during the search process the vertices in decreasing order of their post values from the step 1 (to label vertices with their scc numbers).

Aside: crawling the web. Very similar ideas to those we discussed are used in crawling the web to discover its “nodes” (documents). We collect the links pointing to new documents that need to be explored. IN each step we pick a link that looks “most interesting” and go there (so, not a bfs nor dfs). How do we know the link has not been explored? We keep a set of all documents we have seen and check new documents against that list. How to do it efficiently - by \textit{hashing}.
7 Paths in graphs

The distance between two nodes in a graph is the length of a shortest path between them.

What is the length of a path? For unweighted graphs — the number of edges in the path (possibly 0). For a weighted graph (each edge \((u,v)\) has its length, \(l(u,v)\) or \(l_{uv}\)) — the sum of the lengths of the edges.

We note that in undirected graphs, we must have \(l(u,v) = l(v,u)\). For directed graphs, it might be that \(l(u,v) \neq l(v,u)\) (in fact, it may be that one of these quantities is defined while the other is not).

7.1 Breadth-first search

The idea is to start search in a node and explore first all its neighbors, then the neighbors of the neighbors (that have not been visited yet) and so. Thus, a graph is viewed as consisting of “layers.”

Breadth-first search algorithm implements this idea by means of a queue \(Q\). Initially, \(Q\) contains only \(s\), the one node at distance 0 from \(s\). For each subsequent distance \(d = 1, 2, \ldots\), there is a point in time at which \(Q\) contains all nodes at distance \(d\) and nothing else. As these nodes are processed, the nodes of the next layer, their as-yet-unseen neighbors are injected into the end of the queue.

If we mark the edges through which nodes are discovered, they form the breadth-first search tree. Unlike the DFS tree, all paths from \(s\) in that tree are shortest paths. We call such trees shortest path trees.

```
procedure bfs(G,s)
Input: Graph \(G = (V,E)\), directed or undirected; vertex \(x \in V\)
Output: For all vertices reachable from \(s\), dist\((u)\) is the distance from \(s\) to \(u\)

for all \(u \in V\):
    dist\((u) = \infty\)

dist\((s) = 0\)
Q = [s] Queue containing just \(s\)
```
while $Q$ is not empty:
    $u = \text{eject}(Q)$
    for all edges $(u, v) \in E$:
        if $\text{dist}(v) = \infty$:
            $\text{inject}(Q, v)$
            $\text{dist}(v) = \text{dist}(v) + 1$

To prove correctness of the algorithm, we prove that

For each $d = 0, 1, 2, \ldots$, there is a moment at which (1) all nodes at distance
$\leq d$ from $s$ have their distances correctly set; (2) all other nodes have their
distances set to $\infty$; and $Q$ contain exactly nodes at distance $d$.

The proof is by induction on $d$. The case of $d = 0$ is evident. Let us consider the
moment when the property holds for $d$. When the last of the nodes then on $Q$ is ejected
from $Q$, $Q$ contains only nodes that can be reached from the nodes at exactly distance $d$
by one edge. Hence all of them are at distance $d + 1$ from $s$. For each of them $\text{dist}(u, v)$
is correctly set. Thus, (1) holds. It is also evident that (2) holds, and we already argued that
(3) holds, too.

The running time of the algorithm is $O(|V| + |E|)$ (we assume the adjacency list repre-
sentation). The reason is that each vertex is on $Q$ at most once. Moreover, each edge is
inspected at most once for directed graphs, and at most twice for undirected graphs.

Here we were interested only in distances from $s$. Thus, we do not need to restart the
procedure. If we wanted to visit all nodes according to the BFS method, we would need to
restart as in the case of the DFS method.

### 7.2 Lengths on edges

In most applications, when we are interested in shortest distances in a network of locations,
edges that we can traverse have lengths that in general will be different. For instance,
when you want to drive from San Francisco to Las Vegas, the following graph is the right
abstraction.
In such cases plain BFS cannot be applied directly. We will now consider a method that works when all edges have non-negative lengths and are integers. In such case, we can transform a weighted graph $G$ into an unweighted graph $G'$ by replacing long edges with paths of the appropriate lengths.

Now, to explain the idea behind Dijkstra’s algorithm, we use alarm clocks. We start BFS on the transformed graph and know that for a while not much will be happening. That is, BFS will be spending time moving along the paths that replaced edges and only new vertices will be seen. Thus, we might as well doze off, assuming that at the start, we set up alarm clocks to wake us up, when a real node is discovered. Assuming the situation in the next figure, we set two clocks to go off at time $T_A = 100$ and $T_B = 200$.

Then, we sleep. We wake up at time $T_A (= 100)$ and adjust the clock set for $B$ to $T_B = 150 = T_A + 50$, as we now know of a shorter route to $B$ from $S$ (the one that goes through $A$). If the graph were bigger this would be an estimated arrival time only (as there might be a shorter route still through vertices we still are to discover).

We can now make that idea more precise. The following alarm clock algorithm simulates the execution of BFS on the modified graph $G'$.

- Set an alarm clock for node $s$ to 0
- Repeat until there are no alarms set:
  - Say the next alarm goes off at time $T$, for node $u$. Then:
    - The distance from $s$ to $u$ is $T$
    - For each neighbor $v$ of $u$ in $G$:
      - if there is no alarm yet for $v$, set one for time $T + l(u, v)$
      - if $v$’s alarm is set for later than $T + l(u, v)$, then reset it to this earlier time

This algorithm is almost ready to use, except that we have to have a way to implement a system of alarms. The right data structure for the job is a priority queue. It maintains a set of elements (nodes) with associated numeric keys (alarms times) and supports the following operations:
1. **Insert** — Add a new element to the set

2. **Decrease-key** — Decrease key of a particular element to a new, lower, value (the name is misleading, a better name might be: adjust the queue to reflect a lower key value of an element)

3. **Delete-min** — Return the element with the minimum key and remove it from the set

4. **Make-queue** — Build a priority queue out of the given elements with the given key values

We will discuss several implementations of priority queues later. Now we assume we have an implementation in hand and move on to describing the Dijkstra algorithm formally.

```plaintext
procedure dijkstra(G, l, s)
Input: Graph G = (V, E), directed or undirected; positive edge lengths \( \{l_e | e \in E\} \); vertex s \( \in V \)
Output: For all vertices reachable from s, dist\( (u) \) is the distance from s to u

for all u \( \in V \):
    dist(u) = \( \infty \)
    prev(u) = nil

dist(s) = 0

H = makequeue(V) (using dist values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) \( \in E \):
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```

The array prev stores pointers to nodes from which the best route to a node was discovered. This array allows us to reconstruct the shortest path tree.

Thus, Dijkstra algorithm is simply a simulation of BFS on a modified graph. While that approach does not show it in any easy way, Dijkstra algorithm in the form we presented above works correctly also on graphs with all edges of non-negative lengths. But we will not go into these details here.

What is the running time of Dijkstra's algorithm? It requires making the queue, then \( |V| \) deletemin and \( |V| + |E| \) insert/decreasekey operations. Thus, it all boils down to how fast these operations run in an implementation of a priority queue.
7.3 Priority queue implementations

**Array.** It is the simplest implementation and assumes a universe of objects that potentially can be placed on the queue. In the case of Dijkstra algorithm, this universe consists of all vertices of the graph. To make a priority queue, we simply set all values in the array to ∞. Insert and decreasekey ops are fast as they require only that the key be adjusted. Each op takes $O(1)$ time. The deletemin op requires a scan of the array and takes time $O(n)$ (we also have to make sure there is a way to recognize the element is deleted). Using the array implementation of a priority queue, Dijkstra algorithm runs in time $O(n^2)$.

**Binary heap.** Elements are stored in a complete binary tree, that is, a tree in which every level is filled from left to right and must be full before the next level is started. Moreover, to be a heap, a special ordering constraint is enforced: the key value of any node of the tree is less than or equal to that of its children. A complete tree with $n$ elements and depth $k$ satisfies

$$2^k = 1 + 2^1 + \ldots 2^{k-1} + 1 \leq n$$

thus, $k \leq \log n$.

It follows that the root contains the minimum key element.

To insert, place the element at the bottom of the tree in the first available place and let it “bubble” up by swapping it with its parent as long as necessary. When the process stops we again have a tree that satisfies the heap property stated above. The number of swaps needed is $\log n$. Thus, insert can be implemented to run in $O(\log n)$ time. A decreasekey is similar except that the element is already in the tree.

To deletemin, we return the root value. Then, we have to delete that node. To do it, we take the “last” node in the tree (the rightmost element in the lowest level) and place it in the root of the tree. We then let it sift down by swapping it with its child that has the lower value of the key. Two comparisons are needed per swap, so the time is again $O(\log n)$.

Binary heap can be implemented as a linked structure. However, due to its regularity a much better performance can be achieved by implementing it as an array. Just note that if we rewrite the nodes of the heap into an array from top to bottom and from left to right, then node $i$ has its children in positions $2i$ and $2i + 1$, if both are within the limit $n$, in position $2i$ only, if $2i = n$, and is a leaf otherwise. Similarly, if $i$ is a node and $i \geq 2$, then $[i/2]$ is the parent of $i$.

The makequeue operation can be implemented to run in time $O(n)$ by performing a series of siftdown operations starting with the end of the array (bottom level) and proceeding upwards.

The running time of Dijkstra algorithm with this implementation of the heap is $O(n \log n + (n + m) \log n) = O((n + m) \log n)$. Is it better than the array implementation? Depends on the density of $G$.

**$d$-ary heap.** The same idea as before, only each non-leaf node has fan-out $d$ (except for possibly one node on the last but one level). The parent of node $i$ is $[(i - 1)/d]$ and its children are $\{ (j - 1)d + 2, \ldots \min\{n, jd + 1\}\}$.

If we use pointer representation of the tree, we must have a way to get to the right node in the tree given the node name, not a hard thing to accomplish.
Insert and decreasekey take $O(\log_d n)$ time. Deletemin takes $O(d \log_d n)$
time. Thus, Dijkstra algorithm takes

$$O(nd \log_d n + (n + m) \log_d n)$$

But now we have a freedom to chose $d$. If we choose $d = 1 + \lceil m/n \rceil$, then $d \geq 2$ (assuming $m \neq 0$), and

$$O(nd \log_d n) = O((n + m) \log_d n).$$

Thus, the running time of Dijkstra algorithm now is $O((n + m) \log_d n)$. For sparse graphs ($m = O(n)$), it matches the performance of the binary heap implementation. For dense graphs ($m = \Theta(n^{1+\delta})$, where $0 < \delta \leq 1$), the $\log_d n = \Theta(1)$ and the running time is linear!!

### 7.4 Shortest paths in the presence of negative edges

If negative edges are allowed, considering undirected graphs makes no longer much sense. Shortest paths are simply not defined between vertices in any connected component that contains a negative edge (as we can travel back and forth as long as we want, always making the path shorter). Thus, from now on, we consider only directed graphs.

If some edges have negative lengths, Dijkstra algorithm may not work as it is based in an essential way on the idea that shortest paths from $s$ to any other point $v$ must pass exclusively through nodes that are closer to $v$. But it is not always so,

A crucial feature of Dijkstra algorithm is that the $\text{dist}$ values it maintains are always either overestimations or exact values.

It starts with all of them equal to $\infty$ (an overestimation for reachable nodes, correct value for those non-reachable ones. The array is then modified by setting the value for $s$ to 0 (exact value as no negative cycle though $s$). From that point on, whenever any of these values changes, it is because of the following update along some edge $(u, v) \in E$:

```plaintext
procedure update((u, v) \in E)
\text{dist}(v) = \min\{\text{dist}(v), \text{dist}(u) + l(u, v)\}
```

This update just says that the distance to $v$ cannot be larger than the distance to $u$ plus $l(u, v)$. It has the following properties:

1. It gives the correct distance to $v$ in the case when $u$ is the second to last node in a shortest path to $v$ and $\text{dist}(u)$ is correctly set
2. It provides an upper bound to the correct distance (in other words, extraneous update’s cannot hurt.

Dijkstra algorithm can be viewed as a carefully arranged sequence of updates (along edges we are sure originate in nodes to which the distance is already correct).

We also know this ordering does not work in general. Is there any other ordering that does? Let us look at the shortest path from \( s \) to \( t \), say \( s, u_1, \ldots, u_k, t \). We can assume this path has at most \( n-1 \) edges (otherwise, it would contain a cycle and, as it is the shortest path, the length of the cycle cannot be negative; thus, the cycle can be dropped from the path). If the sequence of updates performed includes a subsequence (not necessarily consecutive) \((s,u_1),(u_1,u_2),\ldots,(u_k,t)\), then, by the two properties, the distance to \( t \) will be correctly computed.

How to ensure that in our order of updates each such sequence shows up as a subsequence? Just perform update’s along all edges \( n-1 \) times. The resulting procedure runs in time \( O(nm) \). It is called the Bellman-Ford procedure.

```
procedure shortest-paths(G,l,s)
Input: Directed graph \( G = (V,E) \); edge lengths \( \{l_e | e \in E\} \) (no negative cycles); vertex \( s \in V \)
Output: For all vertices reachable from \( s \), \( \text{dist}(u) \) is the distance from \( s \) to \( u \)

for all \( u \in V \):
    \( \text{dist}(u) = \infty \)
    \( \text{prev}(u) = \text{nil} \)

\( \text{dist}(s) = 0 \)

repeat \( n-1 \) times:
    for all \( e \in E \)
    \( \text{update}(e) \)
```

If there are no negative cycles in the graph (an assumption we made), shortest paths to all reachable vertices are defined and this method works. But if there are negative cycles, there may be no shortest path from \( s \) to some vertices. We can easily detect the presence of negative cycles. It is enough to run one more round of update’s along all edges and at least for one node its \( \text{dist} \) value will go down.

If we run \( n - 1 \) rounds of update’s and follow it up with another series of \( n - 1 \) update’s, for nodes whose \( \text{dist} \) value has not changed, shortest paths are defined and the \( \text{dist} \) values contain the correct distances.
7.5 Shortest paths in dags

Directed acyclic graphs cannot have negative cycles and so, shortest path problem is well defined for all vertices.

The right ordering for updates is to perform them according to the linearized (topological) order of the graph (correctness follows from the fact that every path in an acyclic graph “respects” that order).

```plaintext
procedure dag-shortest-paths(G,l,s)
Input: Directed acyclic graph G = (V,E); edge lengths {l_e | e ∈ E}; vertex s ∈ V
Output: For all vertices reachable from s, dist(u) is the distance from s to u

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil

dist(s) = 0
Linearize G

for all u ∈ V, in linearized order:
    for all edges (u,v) ∈ E: update(u,v)
```

Note that this algorithm can be used to compute longest paths, too!!
8 Greedy algorithms

Greedy algorithm apply to optimization problems, where optimal solutions can be found by proceeding in an optimal way locally in each step.

We will now discuss several problems that can be solved exactly by means of greedy algorithms.

8.1 Minimum spanning trees

Here is the problem: you are to network a collection of computers by linking selected pairs. You are given a set of potential links from which to choose. Moreover, for each potential link you know the cost of maintaining it. You want to connect the computers into one network by means of links with the lowest total maintenance cost.

Clearly, the optimal solution must not contain a cycle. Indeed, removing any edge would not disconnect the network and would reduce the cost. Thus, what we are after is a minimum spanning tree.

Formal definition follows:

**Input:** An undirected connected graph \( G = (V, E) \); edge weights \( w_e \)

**Output:** A tree \( T = (V, E') \), with \( E' \subseteq E \), that minimizes

\[
\text{weight}(T) = \sum_{e \in E'} w_e.
\]

In our example, the minimum spanning tree has cost 16.

8.2 A greedy approach

Kruskal’s minimum spanning tree algorithm starts with the empty graph and then selects edges from \( E \) according to the following rule:

*Repeatedly add the next lightest edge that does not produce a cycle.*

This is a greedy approach. Every decision is optimal “locally” yet results in a globally optimal solution.
8.3 A few properties of trees

Property 2. A tree with \( n \) nodes has \( n - 1 \) edges.

Property 3. A connected undirected graph with \( n \) vertices and \( n - 1 \) edges is a tree.

Property 4. An undirected graph is a tree if and only if there is a unique path between any pair of nodes.

8.4 The cut property

**Cut property.** Suppose edges \( X \) are part of a minimum spanning tree of \( G = (V, E) \). Pick any nonempty subset of nodes \( S, S \neq V \), for which \( X \) does not cross between \( S \) and \( V - S \), and let \( e \) be the lightest edge across this partition. Then \( X \cup \{e\} \) is part of some MST.

Proof: Let \( T \) be a minimum spanning tree containing \( S \). If \( e \in E(T) \), the claim is obvious. Thus, let \( e \notin E(T) \). There is an edge in \( T \), say \( e' \), that crosses between \( S \) and \( V - S \) (otherwise, \( T \) is not a spanning tree). Moreover, \( e' \neq e \), as \( e \notin E(T) \). Clearly, \( T' = (T - e') + e \) is a spanning tree (is connected, spanning and has \( n - 1 \) edges). Since \( w_e \leq w_{e'} \),

\[
weight(T) \leq weight(T') \leq weight(T).
\]

Thus, \( T' \) is a minimum spanning tree, too. Moreover, \( X \cup \{e\} \subseteq E(T') \).

The cut property implies the correctness of Kruskal’s algorithm. Indeed, the following property holds after each iteration: the set of edges selected is contained in an MST.

To prove it, we proceed by induction. The basis is clear. If the claim holds after \( k \) iterations and we are not done yet, then the edge selected in the iteration \( k + 1 \) crosses a cut (one part consists of the vertices in the connected component of the forest constructed so far; the other part consists of all other vertices).

```plaintext
procedure kruskal(G,w)
Input: A connected undirected graph \( G = (V,E) \); edge weights \{\( w_e | e \in E \)\}
Output: A minimum spanning tree defined by the edges \( X \)

for all \( u \in V \):
    makeset(\( u \))

\( X = \emptyset \)

sort all edges in \( E \) by weight
for all edges \( \{u,v\} \in E \), in increased order of weights:
    if \( \text{find}(u) \neq \text{find}(v) \):
        add edge \( \{u,v\} \) to \( X \)
        union(\( u,v \))

Procedures makeset, find and union are used to manage partitions of the vertex set of the graph: collections of disjoint and non-empty subsets that cover all vertices.
```
makeset\( (x) \) creates a singleton set \( \{x\} \); find\( (x) \) computes the name of the set to which \( x \) belongs; union\( (u, v) \) forms the union of the sets containing \( x \) and \( y \).

### 8.5 A data structure for disjoint sets

We will represent each set as a directed tree defined by parent pointers. The root serves as the id (name) of the set. The root is distinguished from other nodes by the fact that its parent pointer is a self-loop.

In addition to a parent pointer, each node has also a rank (for now, you can think of the rank as the height of the tree)

```
procedure makeset\( (x) \)
\[
\pi(x) = x \\
\text{rank}(x) = 0
\]
```

```
procedure find\( (x) \)
\[
\text{while } x \neq \pi(x): \quad x = \pi(x) \\
\text{return } x
\]
```

Why rank?

```
procedure union\( (x, y) \)
\[
\text{if } \pi(x) = \pi(y): \quad \text{return} \\
\text{if } \text{rank}(\pi(x)) > \text{rank}(\pi(y)): \quad \pi(\pi(y)) = \pi(x) \\
\text{else:} \\
\quad \pi(\pi(x)) = \pi(y) \\
\quad \text{if } \text{rank}(\pi(x)) = \text{rank}(\pi(y)): \quad \text{rank}(\pi(y)) = \text{rank}(\pi(x)) + 1
\]
```

To illustrate, we consider an example in which we have a set \( S = \{A, B, C, D, E, F, G\} \), and execute the following sequence of operations: makeset\( (x) \), for \( x \in S \), and then, union\( (A, D) \), union\( (B, E) \), union\( (C, F) \), union\( (C, G) \), union\( (E, A) \), and union\( (B, G) \).

More generally, using rank for union, allows us to control the height of the resulting trees.

**Property 1.** For any \( x \), rank\( (x) < \text{rank}(\pi(x)) \).

Proof: The rank is designed to be exactly the height of the subtree rooted at that node. Thus, when moving up the tree, the rank values are strictly increasing. \(\square\)
Property 2. Any root node of rank $k$ has at least $2^k$ nodes in its tree.
Proof: A root node with rank $k$ is created by the merger of two trees with roots of rank $k - 1$. We can use this observation to prove the property by induction. □

Corollary 1. Any node of rank $k$ has at least $2^k$ nodes in its tree.
Proof: As long as a node is the root of its tree, the assertion holds by Property 2. At the point when the node stops being the root, its rank and the number of its descendants stop changing. □

Property 3. If there are $n$ elements overall, there are at most $n/2^k$ nodes of rank $k$.
Proof: Property 1 shows that each node has at most one ancestor of rank $k$. If $m$ is the number of nodes of rank $k$, then the number of nodes in their subtrees is at least $m2^k$. Thus, $m2^k \leq n$ and $m \leq n/2^k$. □

Corollary 2. The maximum rank is no more than $\log n$. Thus, the height of all trees at any time is at most $\log n$.
Proof: If $x$ is a node of maximum rank, say $r$, then $n/2^r \geq 1$ (otherwise, we get a contradiction with Property 3). Thus, $r \leq \log n$. □

When rank is used, we can implement the Kruskal’s algorithm so that it runs in time $O(m \log n)$ for sorting plus $O(m \log n)$ for the union and find operations.

But what if edges are given to us in a sorted order? Or, if they are small integers, and so can be sorted quickly?

8.6 We can do better — path compression

The idea is to pay a little upfront to make it possible to keep the data structure in a better shape. The idea is to spend a little extra time at each find so that to make the tree “shorter” and so, easier to process down the road.

```
procedure find(x)
if x ≠ π(x): π(x) = find(π(x))
return π(x)
```

We do not see the benefit of this modification right away. The benefit is long-term rather than immediate. And so, we need a different type of an analysis. We will look at sequences of find and union operations and estimate the average time per operation. This amortized time turns out to be just barely more than $O(1)$ down from $O(\log n)$.

Think of the data structure as having a top level consisting of roots, and below it, insides of the trees. Find operations work on the insides, whereas unions only look at the top level. The path compression has no effect on union operations and leaves the top level unchanged.

In particular, the ranks of root nodes are unchanged. Moreover, once a node ceases to be a root, it never resurfaces as a root and its rank is fixed forever. Therefore, path compression does not affect the rank of any node (although the interpretation of rank(x)
as the height of the tree rooted in \( x \) no longer makes sense). It is easy to see that Properties 1 and 2 still hold. Moreover, as performing path compression has no effect on the number of nodes of rank \( k \), it follows that Property 3 holds, too.

If there are \( n \) elements, their ranks can range from 0 to \( \log n \) (Property 3). We will divide the nonzero part of that range into carefully selected intervals

\[
\{1\}, \{2\}, \{3, 4\}, \{5, \ldots, 16\}, \{17, \ldots, 2^{16} = 65536\}, \{65537, 2^{65537}, \ldots\}
\]

That is, each interval is of the form \( \{k + 1, \ldots, 2^k\} \), where \( k = 0 \), or \( k > 0 \) is a power of 2. The number of groups is \( \log^* n \), the number of successive log operations that need to be applied to \( n \) to bring the value to 1 or below. For instance, \( \log^* 1000 = 4 \) as \( \log \log \log 1000 \geq 1 \) but \( \log \log \log \log 1000 < 1 \). In practice, only the first five (or fewer) of the intervals will show up, as we will never have \( n \geq 2^{65536} \).

We now give each node some pocket money (the total doled out is \( n \log^* n \) dollars) and show that each \texttt{find} takes \( O(\log^* n) \) steps plus some additional time that can be “paid for” by the pocket money of the nodes involved. Thus, the overall time for \( m \) \texttt{find}'s is \( O(m \log^* n + n \log^* n) \).

Specifically, a node receives its allowance as soon as it ceases to be a root, at which point its rank is fixed. If its rank lies in the interval \( \{k + 1, \ldots, 2^k\} \), the node receives \( 2^k \) dollars. By Property 3, the number of nodes with rank > \( k \) is bounded by

\[
\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}.
\]

Therefore the total money given to nodes with the rank fixed to a value in this particular interval is at most \( n \) dollars and, since there are \( \log^* n \) intervals, the total money distributed is \( \geq n \log^* n \).

Now, the time taken by a specific \texttt{find} is simply the number of pointers followed. Consider the ascending rank values along the chain of nodes up to the root. These nodes \( x \) fall into two categories: either the rank of \( \pi(x) \) is in a higher interval than the rank of \( x \), or else it lies in the same interval. There are at most \( \log^* n \) nodes of the first type, so the work done on them takes \( O(\log^* n) \) time. The remaining nodes, whose parents’ ranks are in the same interval as theirs, have to pay a dollar out of their pocket money for their processing time.

But do they have enough money to pay for all the times they are involved in \texttt{find} as nodes of that other type? Here is a crucial observation: each time \( x \) pays a dollar, its parent changes to one of higher rank. Therefore, if \( x \)'s rank is \( n \) the interval \( \{k + 1, \ldots, 2^k\} \), it has to pay no more than \( 2^k \) dollars (its allowance) before its parent’s rank is in a higher interval; whereupon it never has to pay again.

### 8.7 Prim’s algorithm

What the cut property tells us is that the following greedy schema is guaranteed to work.
Prim’s algorithm starts by setting $S$ to some vertex $s$. Then in each step it includes to $S$ the lightest edge between $S$ and outside of $S$. Equivalently, it attaches to $S$ a vertex $v$ outside of $X$ connected to a vertex in $S$ with the lightest edge, that is, having the lowest cost, where

$$\text{cost}(v) = \min_{u \in S} w(u,v).$$

There is a strong resemblance to Dijkstra algorithm and the pseudo-code is almost identical.

**procedure prim$(G,w)$**

Input: Graph $G = (V,E)$, undirected, with edge weights $w_e$

Output: A minimum spanning tree defined by the arrow $\text{prev}$

1. for all $u \in V$:
   - $\text{cost}(u) = \infty$
   - $\text{prev}(u) = \text{nil}$
2. pick any initial node $u_0$ $\text{cost}(u_0) = 0$
3. $H = \text{makequeue}(V)$ (using cost values as keys)
4. while $H$ is not empty:
   - $v = \text{deletemin}(H)$
   - for all edges $\{v, z\} \in E$:
     - if $\text{cost}(z) > w(v,z)$:
       - $\text{cost}(z) = w(v,z)$
       - $\text{prev}(z) = v$
       - $\text{decreasekey}(H,z)$

The running time analysis is the same as for the Dijkstra algorithm. Bottom line: it depends on the implementation of a priority queue.
8.8 Huffman encoding

In the MP3 audio compression scheme, a sound signal is encoded in three steps:

1. It is digitized by sampling at regular intervals, yielding a sequence of real numbers $s_1, \ldots, s_T$ (for stereo, two channels are needed doubling the number of samples). If the sampling rate is 44,100 per second, a 50 minute piece would correspond to $50 \times 60 \times 44,100 \approx 130000000 = 130 \times 10^6$ measurements.

2. Each real value $s_i$ is then quantized: approximated by a nearby number from a fixed finite set $\Gamma$ (selected carefully to exploit human perceptual limitations so that the approximating sequence is indistinguishable from $s_1, \ldots, s_T$.

3. The resulting sequence of length $T$ over the alphabet $\Gamma$ is encoded in binary.

Huffman encoding is used in this last step (a similar application — text-file compression).

Let’s look at the case when $\Gamma = \{A, B, C, D\}$. How to express these symbols in binary? We can use two-bit strings!! Then, 260 megabits are needed in total. Can we do better?

Yes, if we want to use codewords of different lengths and if the symbols from $\Gamma$ show up in the sequence with different frequencies. Say $A$ has frequency 70 million, $B$ — 3 million, $C$ — 20 million, and $D$ — 37 million. Since we want to use different length codewords, we must be careful. Can we translate back? How do we know when a codeword ends? If the codewords are $\{0, 01, 11, 001\}$ and we see 0 as the first character and the next two are 1’s, what do we do? Is the first 0 an $A$ followed by $C$, or do the first two characters represent $B$?

We will avoid the problem if the set of codewords is prefix-free: no word is a prefix of another one in the set. Then, whenever we get a chunk of bits forming a codeword, we know it is the time to translate it.

The following set is prefix-free: $\{0, 100, 101, 1\}$. Such sets can be represented as binary trees with codewords as leaves, defined by the corresponding paths (turn left means 0, turn right means 1).

![Binary Tree Diagram]

OK, so now we know how to decode unambiguously. How good is this encoding? Now, the total length of the binary encoding is (in millions of bits)

$$70 \times 1 + 3 \times 3 + 20 \times 3 + 37 \times 2 = 213,$$

64
a 17% improvement! Thus, if an encoding is designed for a particular file (with particular frequencies), we can assign these frequencies to the leaves of the tree and use the following measure of its quality:

\[ \sum_{i=1}^{n} f_i \cdot (\text{depth of the } i\text{th leaf}) \]

But it can also be expressed differently. Let's assign to each internal node the sum of the frequencies of its descendant leaves. Then the cost of the tree is the sum of the frequencies of all leaves and internal nodes except the root (we assume that the root is not the leaf, that would mean that \( \Gamma \) has just one element and still better compression would be possible). Indeed, when decoding, we touch each internal node other than the root and each leaf exactly as many times as the frequency. Moreover, for each touch, one bit is read.

So, how to find the best (optimal) tree? The first formulation of the cost tells us that in the optimum tree two symbols with the smallest frequencies must be at the bottom of the tree (as children of the lowest internal node, which must have two children, optimal code trees are full). Otherwise, we could get a better tree by swapping.

This suggests a greedy approach. Find the two symbols with the lowest frequencies, say \( i \) and \( j \), and make them children of a new node, which has frequency \( f_i + f_j \). To keep the notation simple, let's just assume these are \( f_1 \) and \( f_2 \). By the second formulation, any tree in which \( f_1 \) and \( f_2 \) are sibling leaves has cost \( f_1 + f_2 \) plus the cost for a tree with \( n-1 \) leaves of frequencies \( (f_1 + f_2), f_3, \ldots, f_n \). Thus, looping now produces an optimal tree. Here is an algorithm.

\[
\text{procedure huffman}(f) \\
\text{Input: An array } f[1..n] \text{ of frequencies} \\
\text{Output: An encoding tree with } n \text{ leaves} \\
\text{H} = \text{makequeue}(f) \\
\text{for } k = n + 1 \text{ to } 2n - 1: \\
\quad i = \text{deletemin}(H) \\
\quad j = \text{deletemin}(H) \\
\quad \text{create a node numbered } k \text{ with children } i \text{ and } j \\
\quad f[k] = f[i] + f[j] \\
\quad \text{insert}(H, k)
\]

Another application of priority queue structure. Running time: \( O(n \log n) \).

9 Dynamic programming

Methods such as divide-and-conquer and greedy choice work only for some specific types of problems. Dynamic programming is a much more general algorithm design paradigm.
Dynamic programming solves a problem by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until the whole lot of them is solved, including the one we care about.

Let’s look at the problem of finding a shortest path in a dag to a given vertex. The trick is to linearize the dag.

\[
\text{initialize all dist(·) values to } \infty \\
dist(s) = 0 \\
\text{for each } v \in V \setminus \{s\} \text{ in linearized order:} \\
\quad \text{dist}(v) = \min_{(u,v) \in E} \{\text{dist}(u) + l(u,v)\}
\]

In dynamic programming, in most cases we are not given a dag; it is implicit. Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems: if to solve subproblem \( B \) we need the answer to subproblem \( A \), then there is a (conceptual) edge from \( A \) to \( B \). In this case, \( A \) is thought of as a smaller subproblem than \( B \), and it will always be smaller, in an obvious sense. Here is a non-graph example.

### 9.1 Longest increasing example

A subsequence of a sequence, is a sequence of elements selected from the original sequence, according to the order given by that original sequence. For instance, \((5, 2, 8, 6, 3, 6, 9, 7)\) is a subsequence of \((5, 2, 8, 6, 3, 6, 9, 7)\). We will now solve the problem to find a longest increasing subsequence in a sequence. In the sequence above, one solution is \((2, 3, 4, 7)\), another one is \((2, 3, 4, 9)\).

The space of solutions has here a nice graphical representation. We treat the positions of the sequence as vertices of the graph. We connect a node \( i \) to \( j \) if \( i < j \) and \( a_i \leq a_j \). Edges defined in this way represent legal steps when constructing a longest sequence. Clearly, to have a longest increasing sequence ending at 8, we need to know a longest increasing sequence ending at 6, 5, 4, 2, or 1. And the same pattern as before emerges. There is a dag behind the problem and, even more in this case, the problem itself is that of finding a longest path (its length). The heart of the code follows:
for \( j = 1, 2, \ldots, n \):
\[ L(j) = 1 + \max \{ L(i) : (i, j) \in E \} \]
return \( \max_j L(j) \)

The optimal subsequence can be obtained by careful bookkeeping. We have to keep track of nodes through which updates are made.

Why is divide-and-conquer dynamic programming but not vice versa? Divide-and-conquer solves a large problem by means of solving some small ones (as in logic programming). However, as the subproblems are in some sense disjoint, there are few of them and no work is repeated. So, the approach leads to efficient algorithms (recursive and non-recursive).

On the other hand, dynamic approach does not assume anything about the subproblems. They may involve work that is done over and over again. Thus, recursion does not work well (vide Fibonacci numbers, or even the longest increasing subsequence, too). What works is memoization, or (as we saw above) an appropriate ordering of the subproblems.

9.2 Chain matrix multiplication

The problem: Find an optimal way to compute the product of \( n \) arrays \( A_1 \times A_2 \times \ldots \times A_n \).

For instance, let \( n = 4 \) and \( A_1, \ldots, A_4 \) be \( 50 \times 20 \), \( 20 \times 1 \), \( 1 \times 10 \) and \( 10 \times 100 \), respectively.

What is the right order (distribution of parentheses)? It does make a difference!! \( A_1 \times ((A_2 \times A_3) \times A_4) \) requires 120,200 multiplications, \((A_1 \times A_2) \times (A_3 \times A_4) \) requires only 7,000 multiplications.

As always, we just compute the cost. Careful bookkeeping will give us the order. We assume that we are given a sequence \( m_0, m_1, \ldots, m_n \) that specifies the dimensions of the \( n \) arrays to be multiplied (the dimensions of the array \( A_i \) are \( m_i - 1 \times m_i \)). Let us also assume that the (optimal) cost of multiplying \( A_i \times A_{i+1} \times \ldots \times A_j \) is \( C(i, j) \). Then, the key formula is:

\[ C(i, j) = \min_{i \leq k < j} \{ C(i, k) + C(k + 1, j) + m_{i-1}m_km_j \} \]

It leads to the following code:

for \( i = 1, 2, \ldots, n - 1 \):
\[ C(i, i) := 0 \]
for \( d = 1, \ldots, n - 1 \):
\[ \text{for } i = 1, \ldots, n - d: \]
\[ C(i, i + d) = \min_{i \leq k < i + d} \{ C(i, k) + C(k + 1, j) + m_{i-1}m_km_j \} \]
return \( C(1, n) \)

The running time is \( O(n^3) \).
9.3 All-pairs shortest paths

We assume no negative-length cycles, but negative-length edges are possible. Bellman-Ford algorithm iterated \( n \) times (once for each vertex), yields a \( O(n^2 m) \) algorithm. But we can do better.

We assume the vertices are 1, 2, \ldots, \( n \). We define the subproblem as consisting of computing \( \text{dist}(i, j, k) \), which is the length of the shortest path from \( i \) to \( j \) with intermediate vertices in \( \{1, \ldots, k\} \). Then, we have the following relation:

\[
\text{dist}(i, j, k) = \min\{\text{dist}(i, j, k_1), \text{dist}(i, k, k_1) + \text{dist}(k, j, k_1)\}.
\]

It gives us the following code

\[
\begin{align*}
\text{for } i = 1 \text{ to } n: \\
\quad \text{for } j = 1 \text{ to } n: \\
\quad \quad \text{dist}(i, j, 0) = \infty \\
\text{for all } (i, j) \in E: \\
\quad \text{dist}(i, j, 0) = l(i, j) \\
\text{for } k = 1 \text{ to } n: \\
\quad \text{for } i = 1 \text{ to } n: \\
\quad \quad \text{for } j = 1 \text{ to } n: \\
\quad \quad \quad \text{dist}(i, j, k) = \min\{\text{dist}(i, k, k_1) + \text{dist}(k, j, k_1), \text{dist}(i, j, k_1)\}
\end{align*}
\]

The running time is \( O(n^3) \).