COMPUTER ALGORITHMS

Athasit Surarerks

Introduction

EUCLID's GAME

Two players move in turn. On each move, a player has to write on the board a positive integer equal to the different from two numbers already in the board; this number must be new. The player who cannot move loses the game.

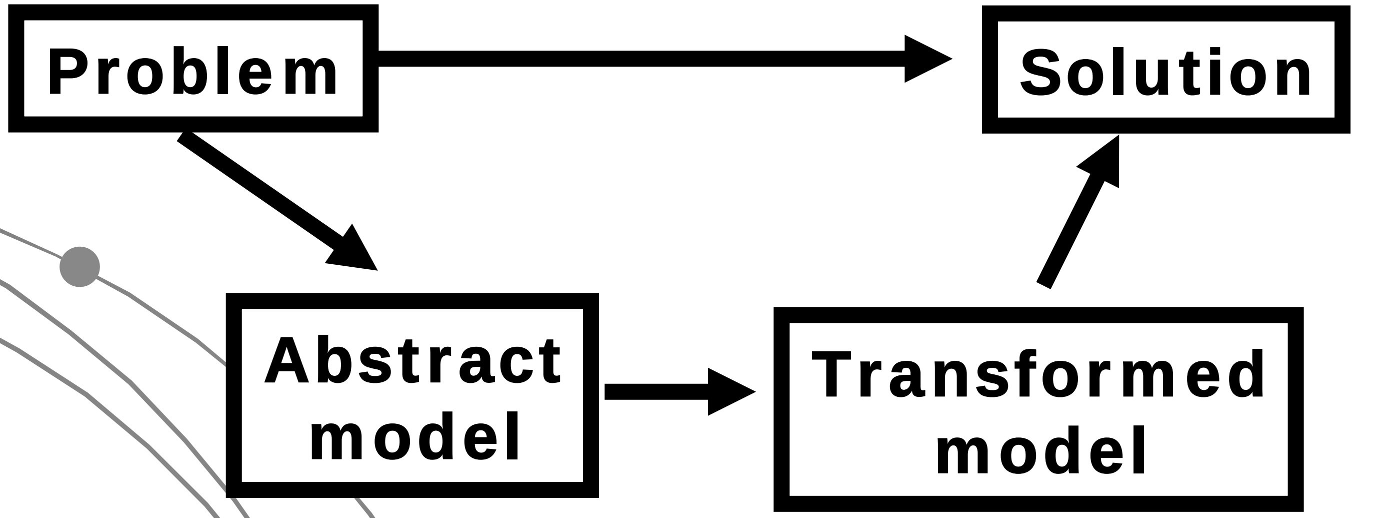
ADDITION's GAME

Player I and II alternately choose integers, each choice being one of the integers 1, 2, …, k, and each choice being made with knowledge of all preceding choices. As soon as the sum of the chosen integers exceeds N, the last player to choose loses.

WORLD PUZZLE

Four people want to cross a bridge; they are all begin in the same side. You must have 17 minutes to get them all across to the other side. It is night, and they have one flashlight. A maximum of two people can cross the bridge at one time. Any party that crosses must have the flashlight with them. Each person walks at a different speed: 1, 2, 5, and 10. A pair must walk together at the rate of the slower person.

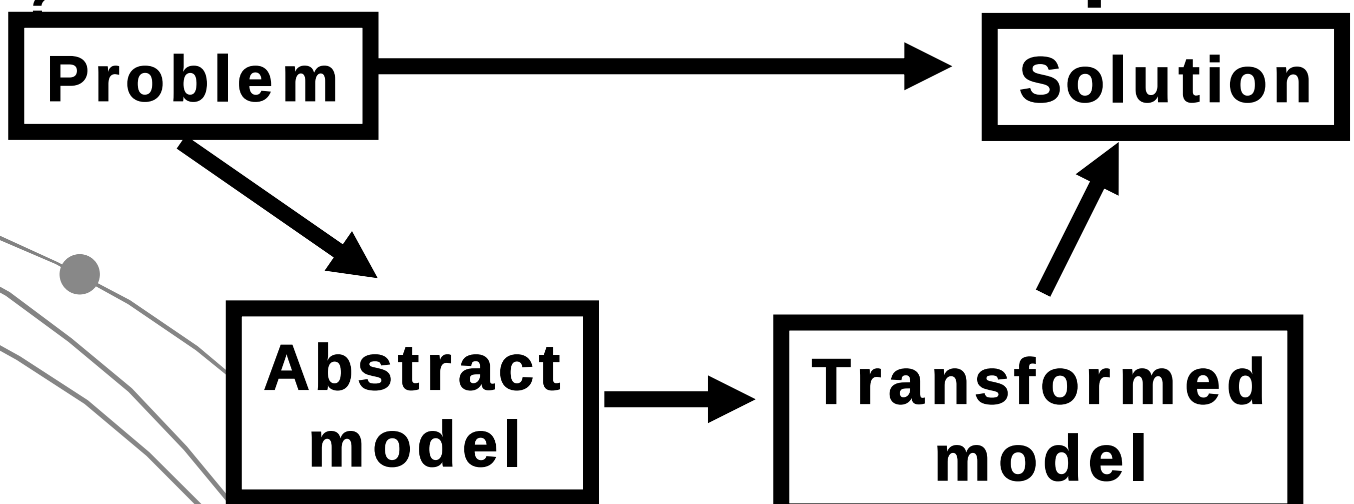
Problem solving



Problem solving

Buy 10 pen (5$ each), sell 8$, at least pen have to sell ?

7 pens



Min β **(** $8\beta \geq 50$ **)**

$\beta = 7$

Complex number

Given two complex numbers,

X = *a* + *b* j

Y = *c* + *d* j

Find an algorithm to perform the product XY.

Cost of a multiplier is 1 USD.
Cost of an adder is 0.01 USD.

4.02 USD

$$(ac\text{-}bd) + (ad+bc)\ \mathrm{j}$$

Complex number

Given two complex numbers,

X = $a + b$ j

Y = $c + d$ j

Find an algorithm to perform the product XY.

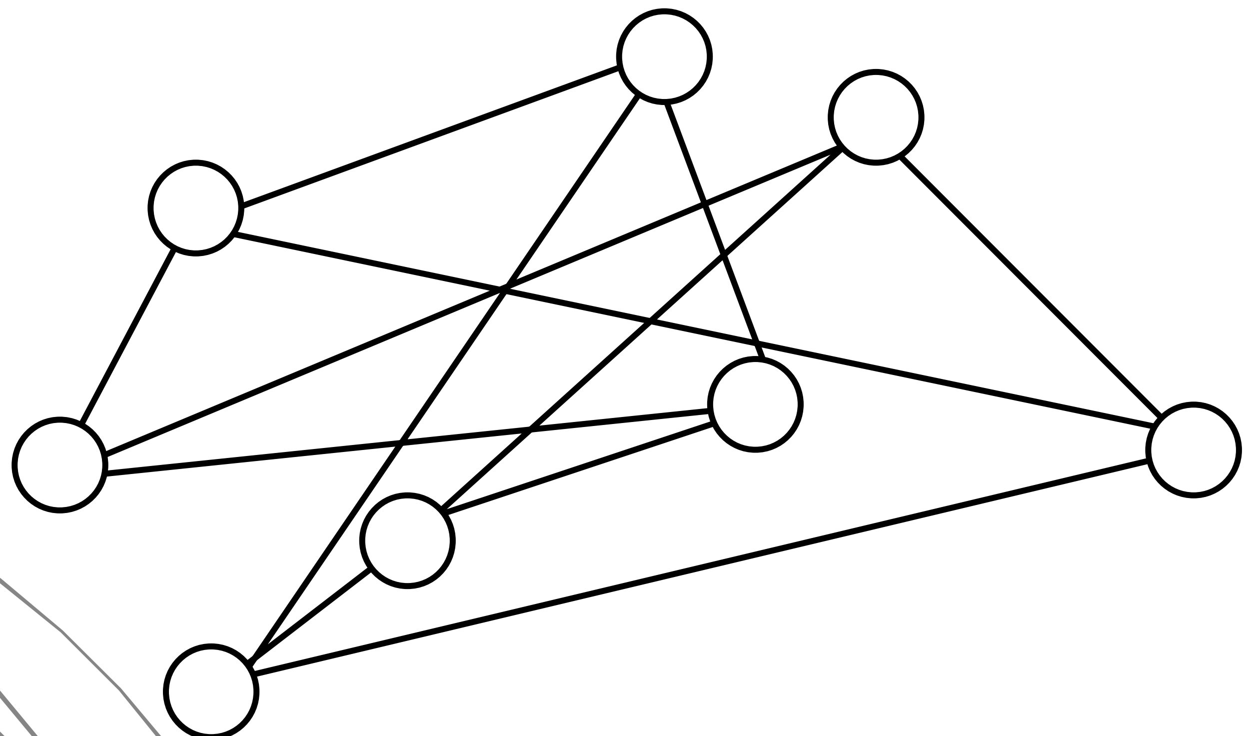
Cost of a multiplier is 1 USD.

Cost of an adder is 0.01 USD.

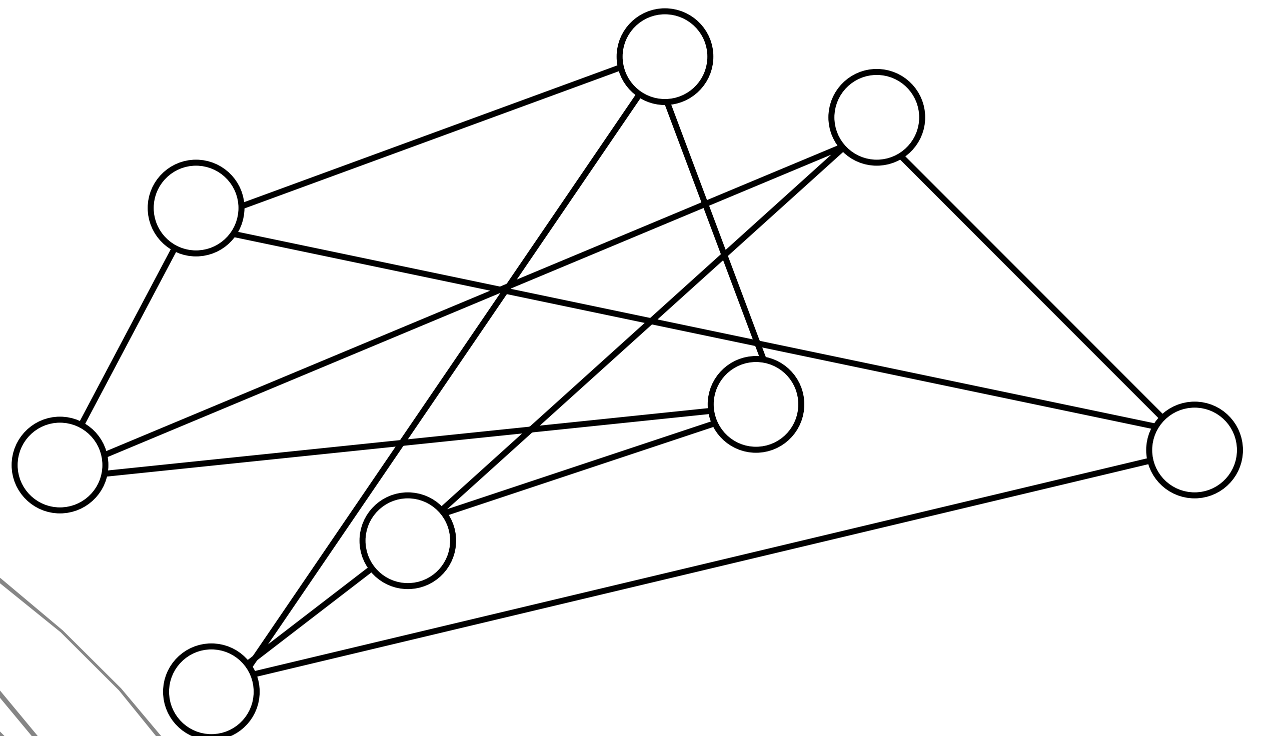
$\boxed{3.05 \text{ USD}}$

$$(a+b)(c+d) = ac+ad+bc+bd$$

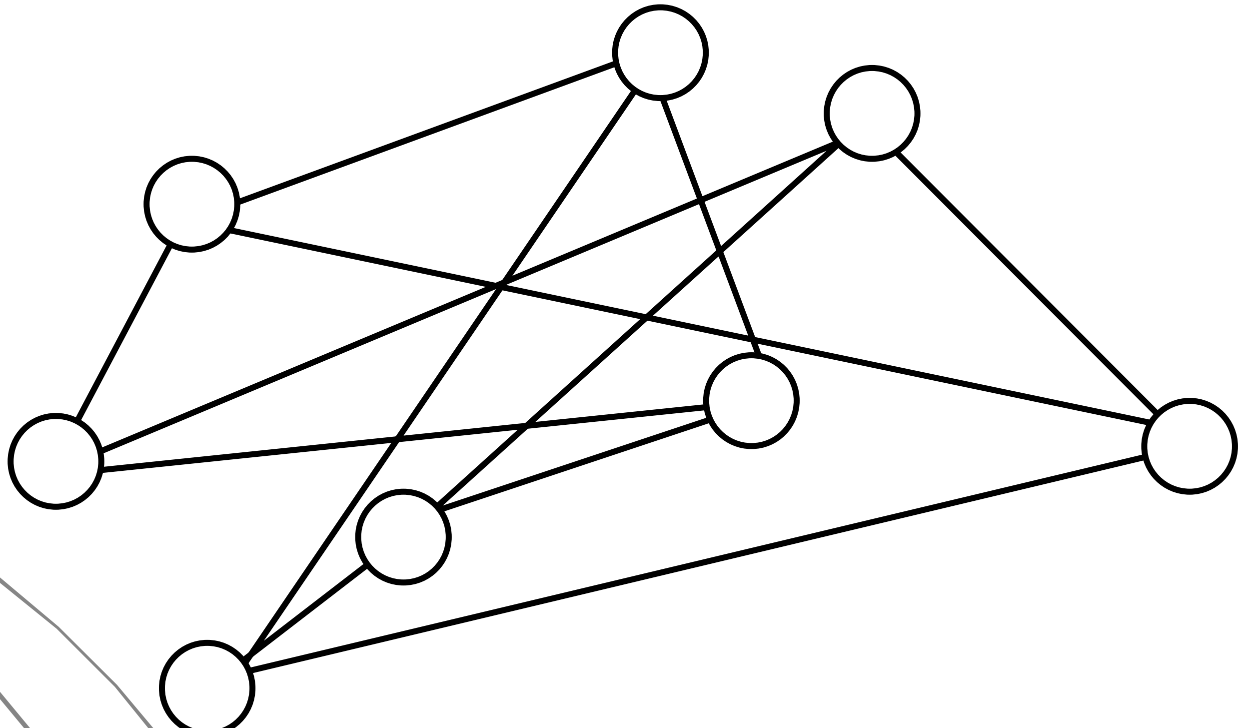
Graph coloring



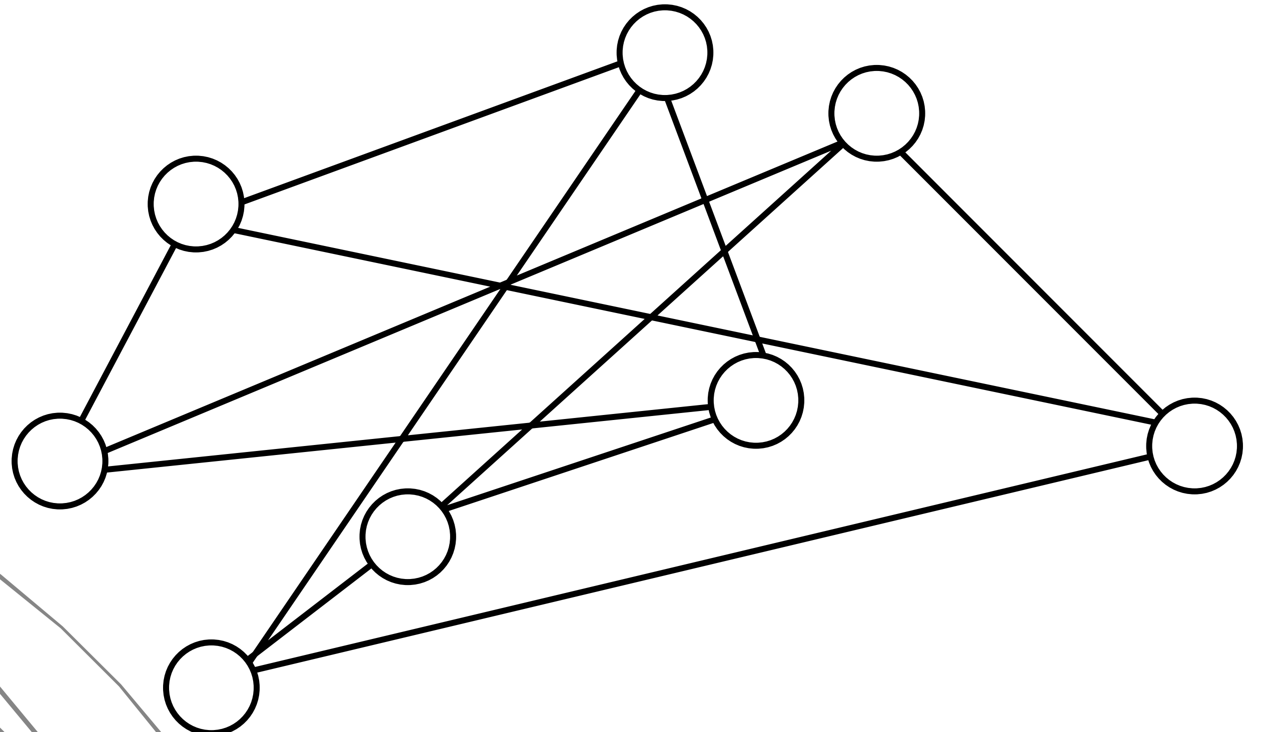
Graph coloring



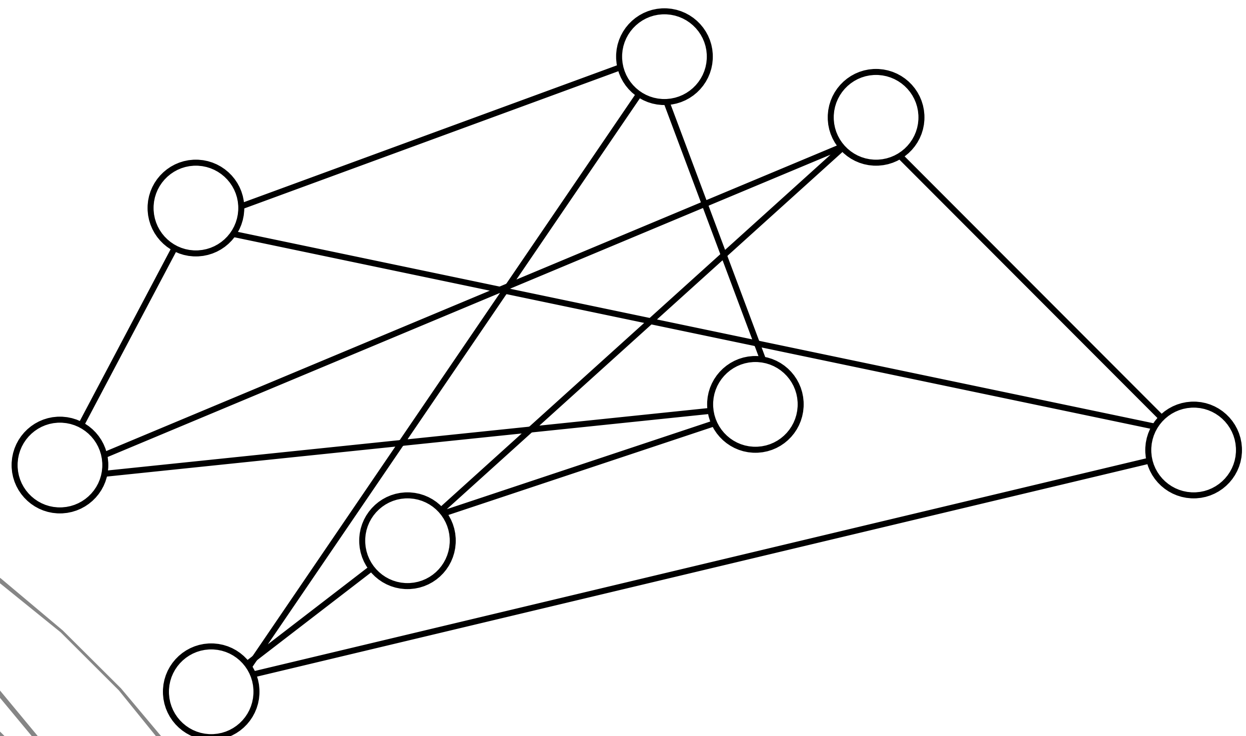
Graph coloring



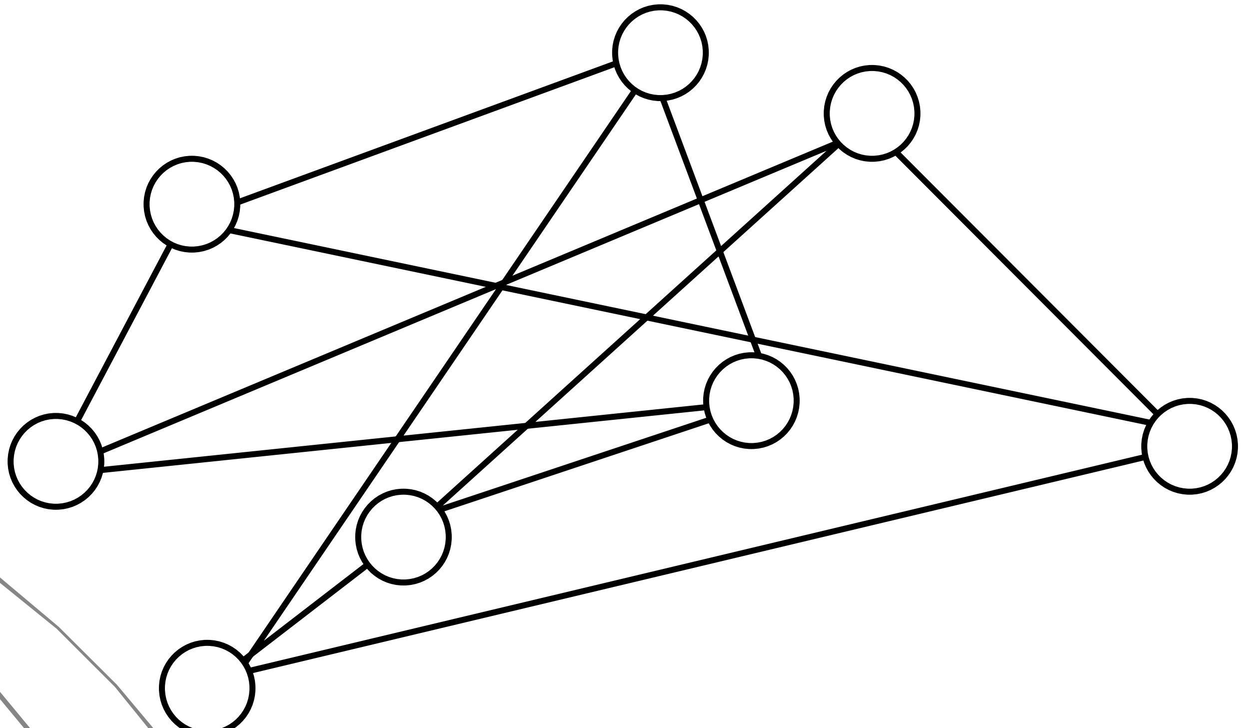
Graph coloring



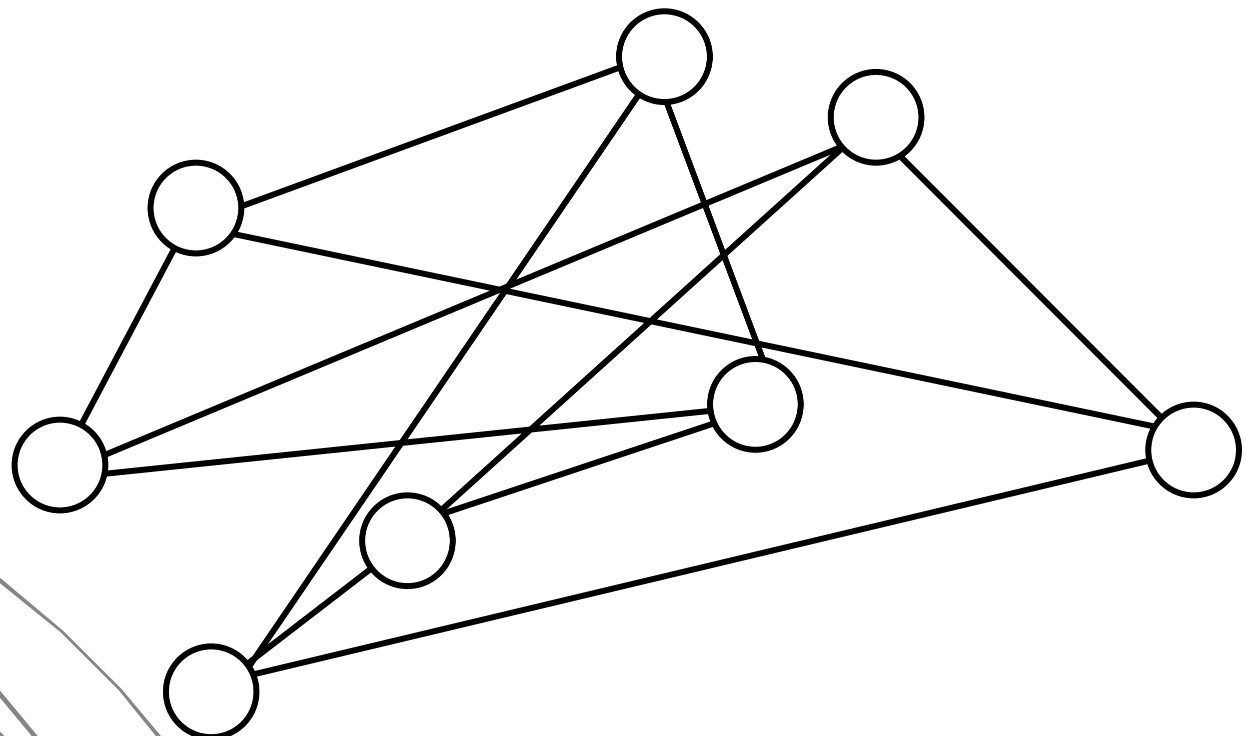
Graph coloring



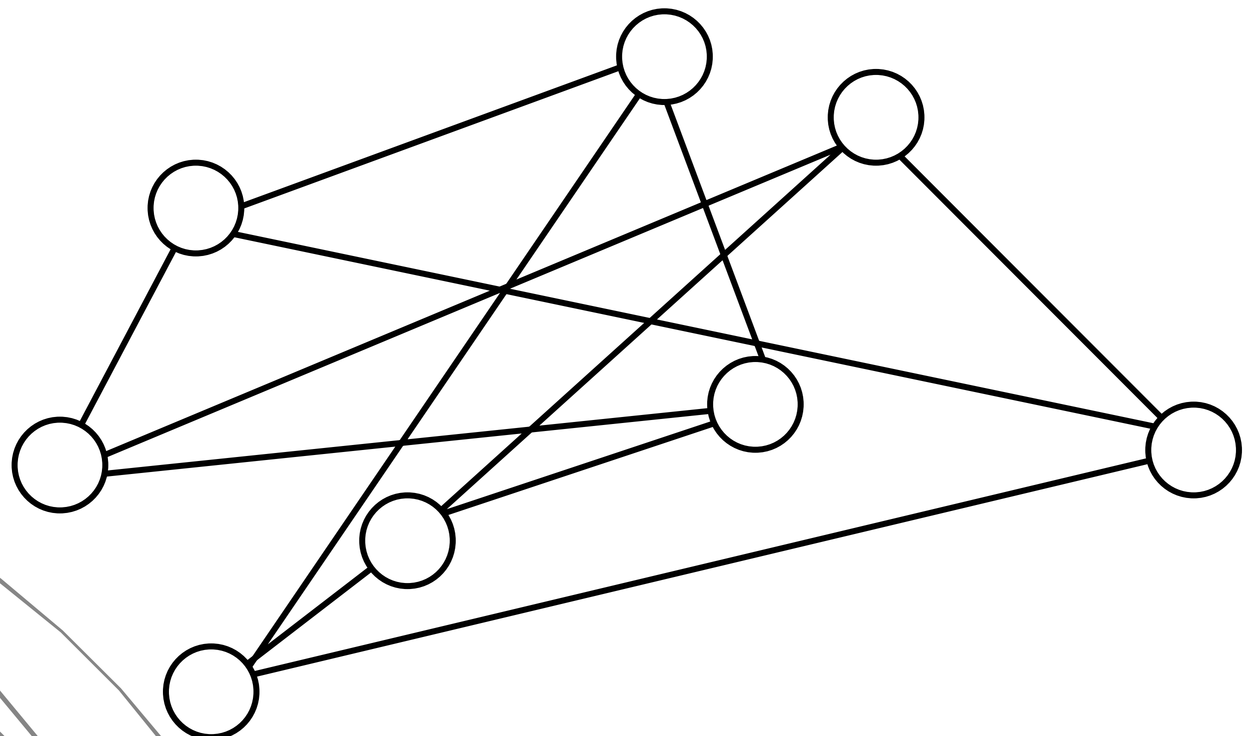
Graph coloring



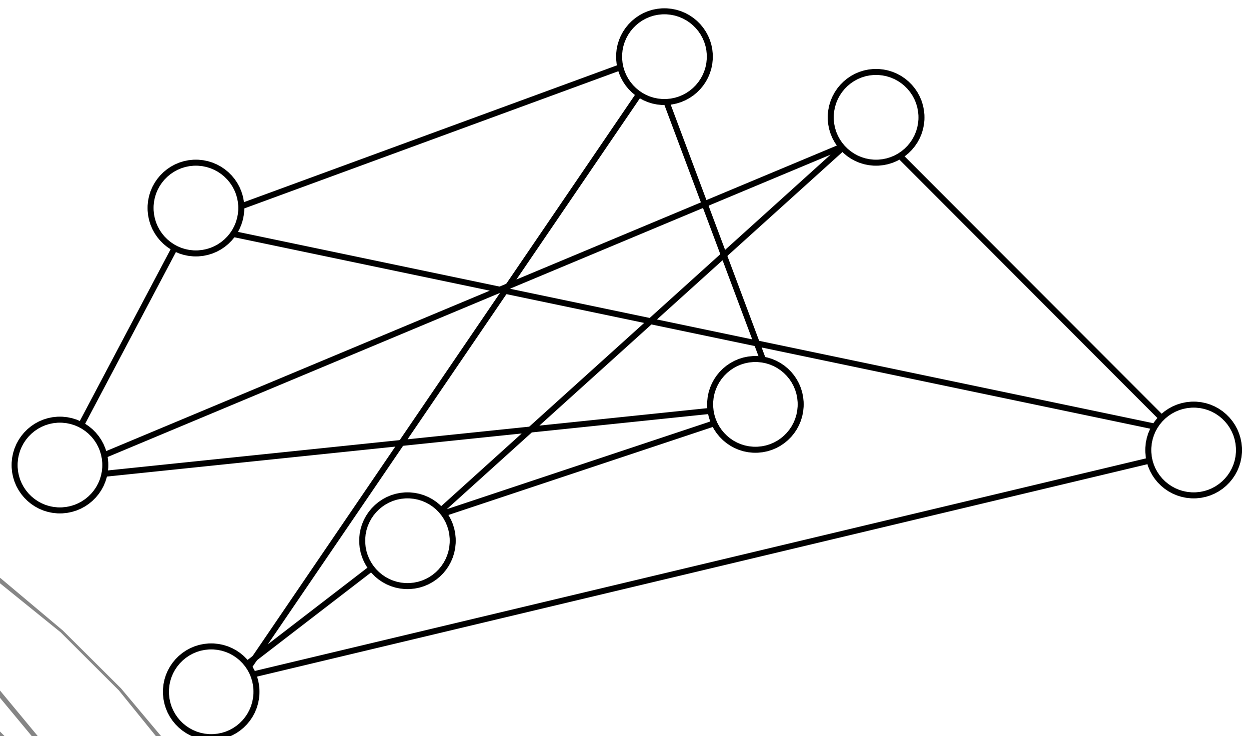
Graph coloring



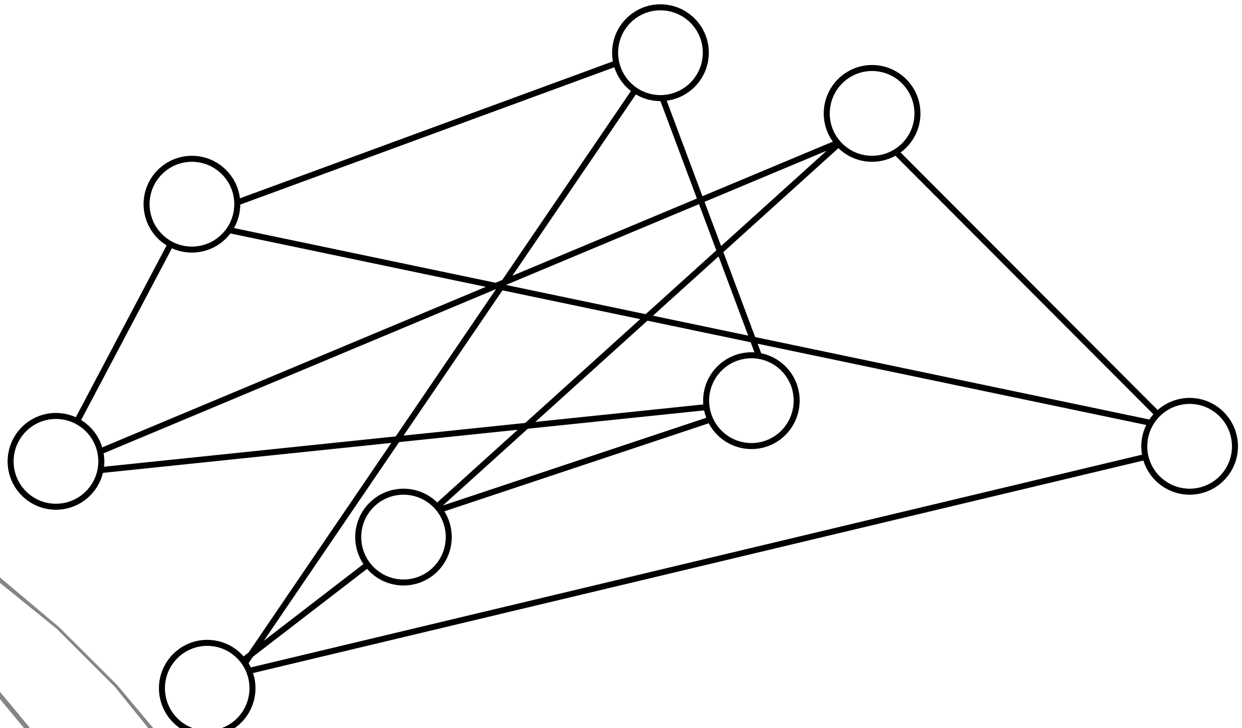
Graph coloring



Graph coloring



Graph coloring



Chromatic number is 2.

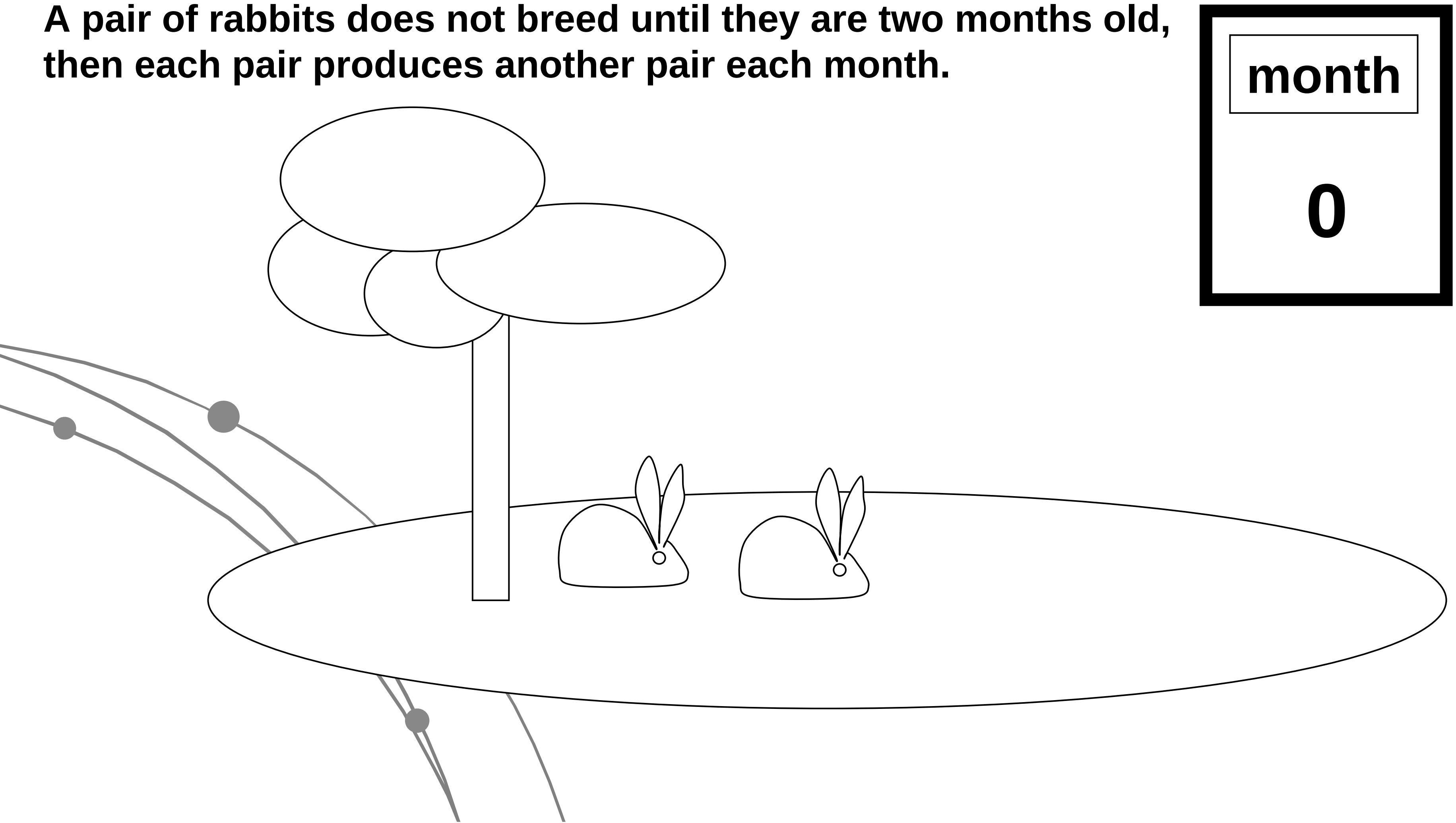
Problem solving

Communication
Language translation
Function
Composition function
Algorithms

Rabbits on an island

By Leonardo di Pisa, 13th century

A pair of rabbits does not breed until they are two months old, then each pair produces another pair each month.



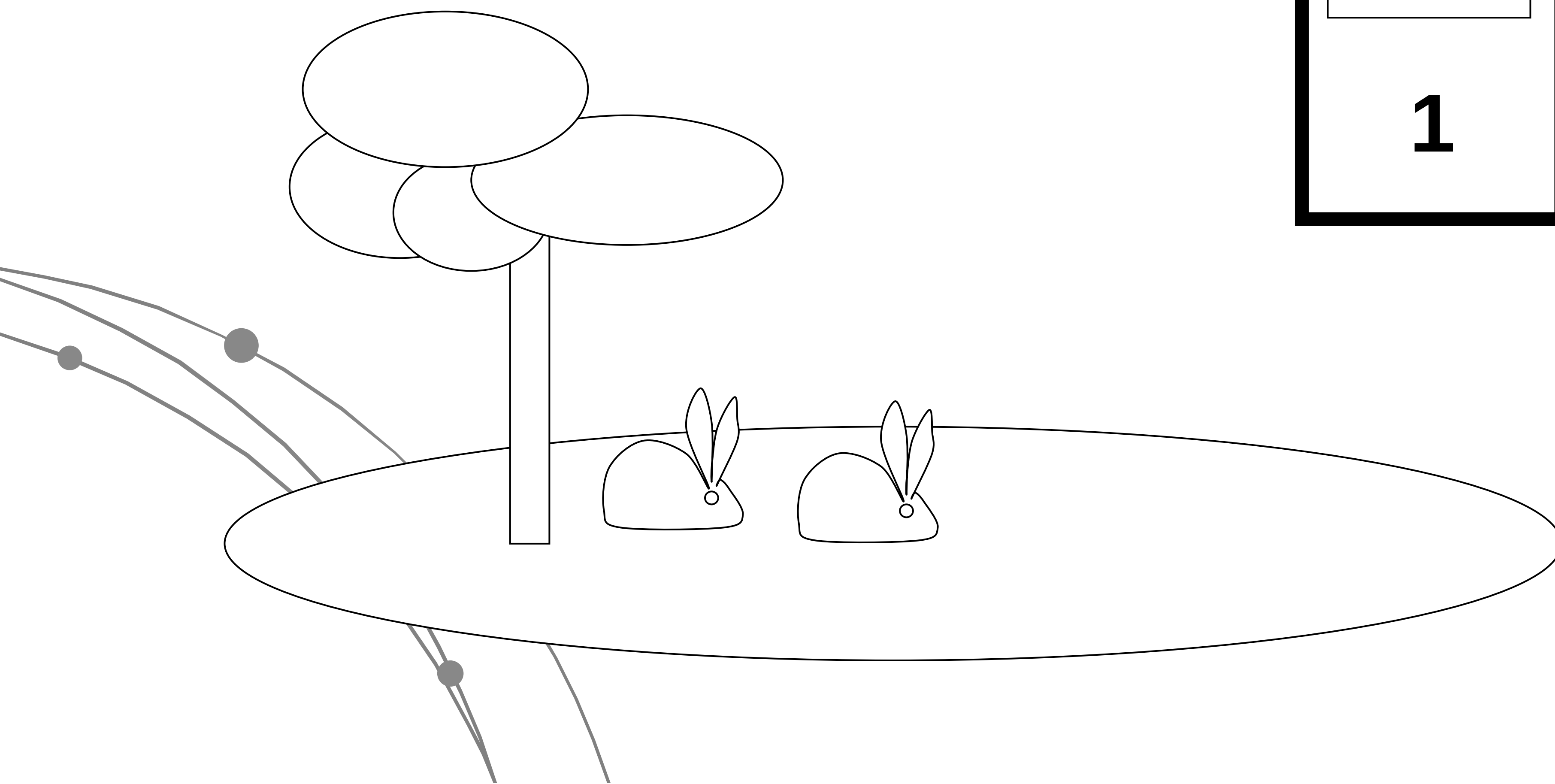
Rabbits on an island

By Leonardo di Pisa, 13th century

A pair of rabbits does not breed until they are two months old, then each pair produces another pair each month.

month

1



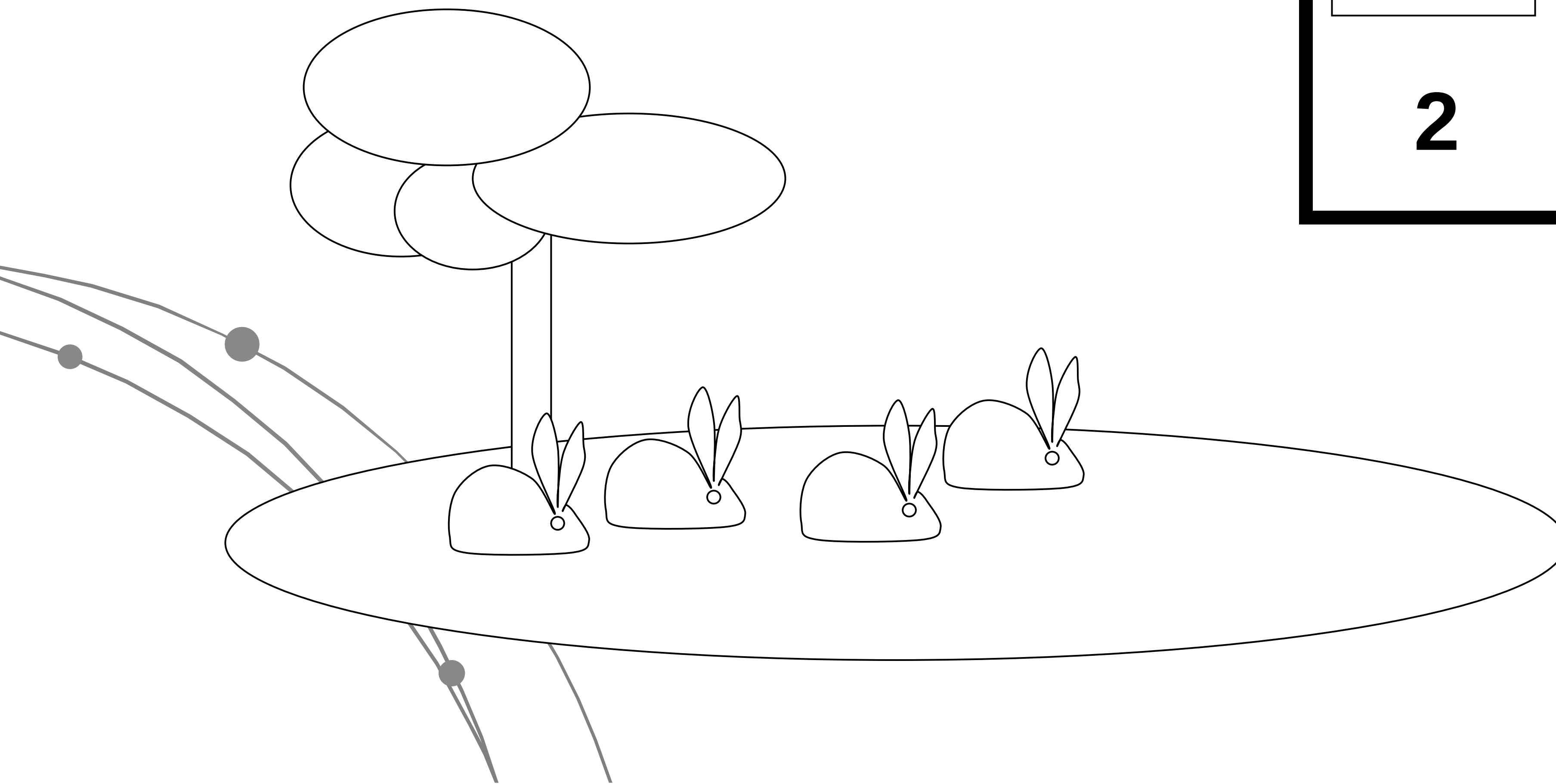
Rabbits on an island

By Leonardo di Pisa, 13th century

A pair of rabbits does not breed until they are two months old, then each pair produces another pair each month.

month

2



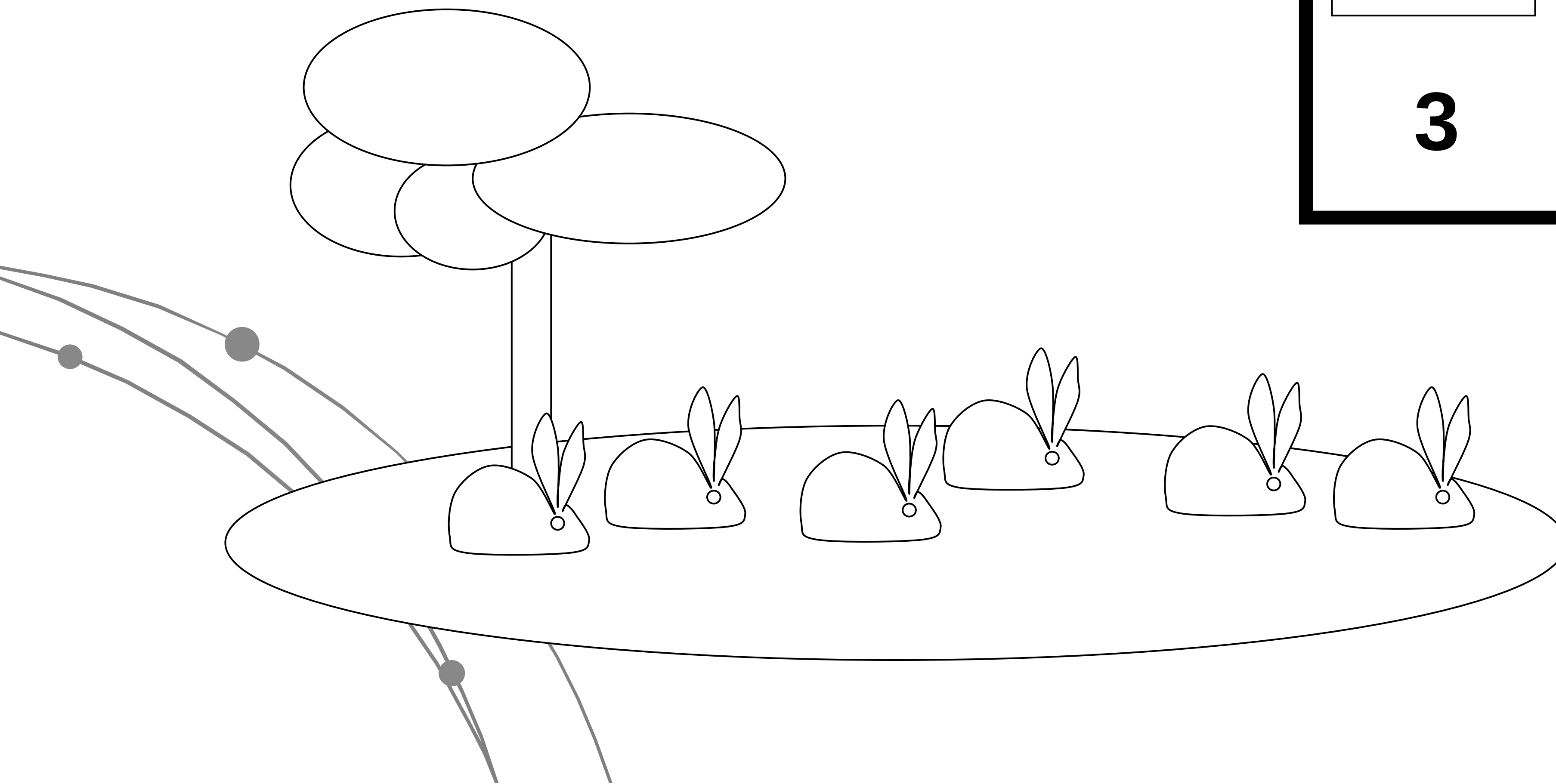
Rabbits on an island

By Leonardo di Pisa, 13th century

A pair of rabbits does not breed until they are two months old, then each pair produces another pair each month.

month

3



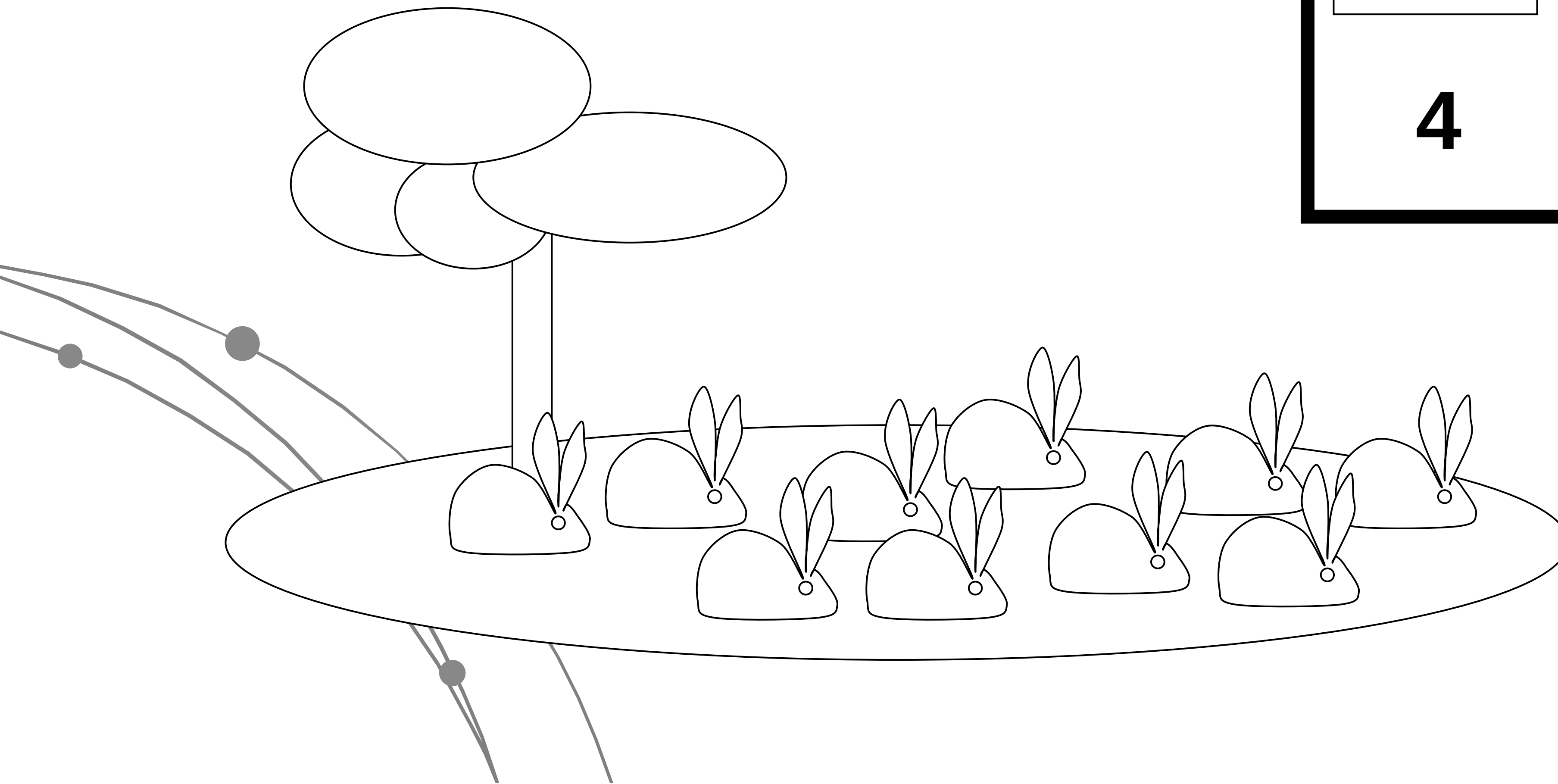
Rabbits on an island

By Leonardo di Pisa, 13th century

A pair of rabbits does not breed until they are two months old, then each pair produces another pair each month.

month

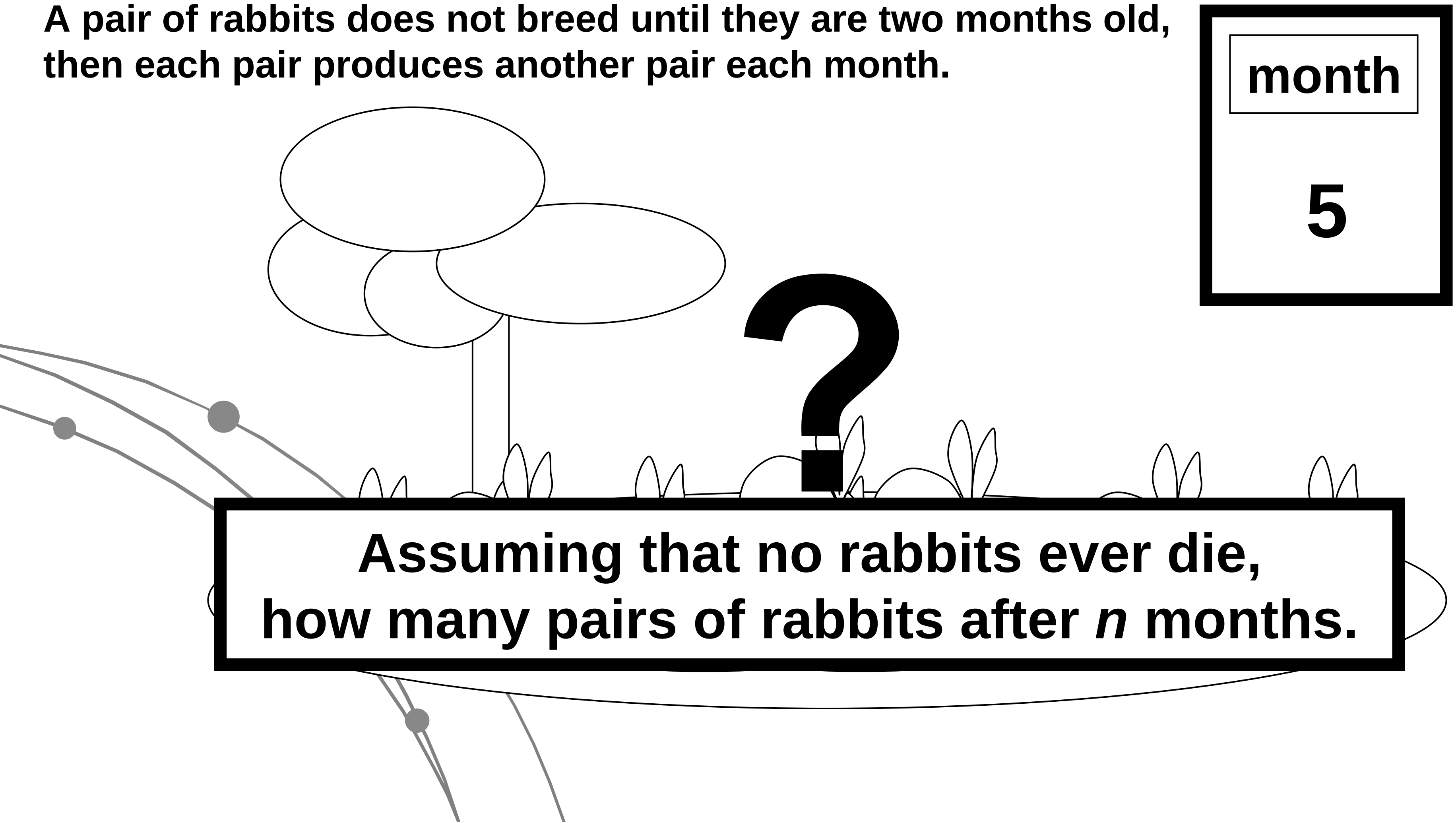
4



Rabbits on an island

By Leonardo di Pisa, 13th century

A pair of rabbits does not breed until they are two months old, then each pair produces another pair each month.



Assuming that no rabbits ever die, how many pairs of rabbits after *n* months.

Problem size

Traveling salesperson

Minimum spanning tree

String matching

Independent set

Sum of subset

Linear partition

Organization

Introduction

Analysis of algorithm

Design of algorithm

Complexity of algorithm

Organization

Introduction

Problems

Asymptotic notations

Organization

Introduction

Analysis of algorithm

Worst-case analysis

Average-case analysis

Amortized analysis

Organization

Introduction

Analysis of algorithm

Design of algorithm

- Divide-and-conquer
- Dynamic programming
- Approximation algorithm

Organization

Introduction

Analysis of algorithm

Design of algorithm

Complexity of algorithm

ALGOL statements

program-name(*variables listed*)

begin *statement(s)* end

variable $\leftarrow$ *expression*

if *condition* then *statement(s)* else *statement(s)* endif

while *condition* do *statement(s)* enddo

for *variable* $\leftarrow$ *initial-value* step *size* until *final-value* do *statement(s)* enddo

goto *label*

return *variables listed*

input *variables listed*

output *variables listed*

any other miscellaneous statement

Maximum-Minimum search

```
001     begin
002     answer ← A[1]
003             for  I ← 2 step 1 until n do
004                     if A[I] > answer then answer = A[I] endif
005             enddo
006     return answer
007     end
```

Time used : $TA_1(n) = t_{002} + nt_{003} + (n\text{-}1)t_{004} + t_{006}$.

Maximum-Minimum search

```
011     max-search (A[1..n])
012     begin
013         if n = 1 then return A[1] endif
014         answer ← max-search(A[1..n-1])
015         if A[n] > answer then return A[n] else return answer endif
016     end
```

Time used : $T_{A2}(n) = T_{A2}(n\text{-}1) + t_{013} + t_{014} + t_{015} = nt_{013} + (n\text{-}1)(t_{014}+t_{015})$.

Efficiency

The efficiency of an algorithm is the resources used to find an answer. It is usually measured in terms of the theoretical computations, such as comparisons or data moves, the memory used, the number of messages passed, the number of disk accesses, etc.

Efficiency

Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of problem. We would like to associate with a problem and integer, called the size of the problem, which is measure of the quantity of input data.

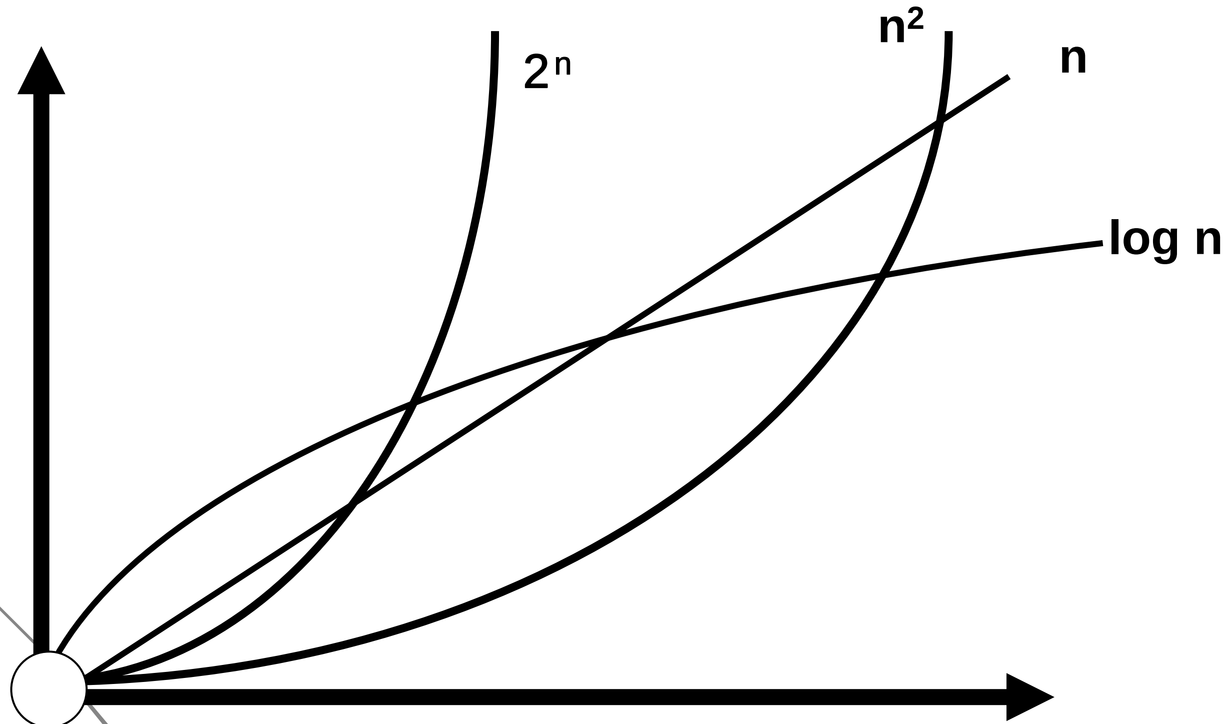
Efficiency

The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. The limiting behavior of the complexity as size increases is called the asymptotic time complexity. Analogous definitions can be made for space complexity and asymptotic space complexity.

Complexity

Algorithm	Growth rate	Time complexity
A_1	$1000n$	n
A_2	$100n\log n$	$n \log n$
A_3	$10n^2$	n^2
A_4	n^3	n^3
A_5	2^n	2^n

Growth rate



Growth rate

Let f and g be two functions, and if

$$f(n) < g(n)$$

if and only if,

$$\lim_{n\to\infty} f(n)/g(n) = 0.$$

Exercises

Compare the growth rate of

- 0.5^n
- 1
- $\log_{10} n$
- n
- 10^n

$$\boxed{0.5^n < 1 < \log n < n < 10^n}$$

Exercises

- Compare the growth rate of
  - $\ln^9 n$ and $n^{0.1}$,
  - n^{100} and 2^n.

L'hÔpital's rules

Let f and g be two functions, and if

$$\lim_{n\to\infty} f(n) = \infty,$$
$$\lim_{n\to\infty} g(n) = \infty$$

Then $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} f(n)'/g(n)'$.

Asymptotic notations

It is the asymptotic complexity of an algorithm which ultimately determines the size of problems that can be solved by the algorithm.

ω

A theoretical measure of the execution of an algorithm, usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation $f(n) = (g(n))$ means $g(n)$ becomes insignificant relative to $f(n)$ as n goes to infinity. More formally, it means that for any positive constant c, there exists a constant k, such that $0 \leq cg(n) < f(n)$ for all $n \geq k$. The value of k must not depend on n, but may depend on c. That is

$$\boxed{\omega(g(n)) = \{f(n) \mid \lim_{n\to\infty}(f(n)/g(n))=\infty \}.}$$

Θ

A theoretical measure of the execution of an algorithm, usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation $f(n) = \Theta(g(n))$ means it is within a constant multiple of $g(n)$. More formally, it means there are positive constants c_1, c_2, and k, such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

$$\boxed{\Theta(g(n)) = \{f(n) \mid \lim_{n\to\infty}(f(n)/g(n))=c, c\neq 0, c\neq\infty\}.}$$

for all $n \geq k$. The values of c_1, c_2, and k must be fixed for the function f and must not depend on n.

Big-O

A theoretical measure of the execution of an algorithm, usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation $f(n) = O(g(n))$ means it is less than some constant multiple of $g(n)$. More formally it means there are positive constants c and k, such that $0 \leq f(n) \leq cg(n)$

$$\boxed{\mathbf{O(g(n)) = \{f(n) \mid \lim_{n\to\infty}(f(n)/g(n))=c, c\neq\infty\}.}}$$

$$\boxed{O(g(n)) = o(g(n)) \cup \Theta(g(n)).}$$

for all $n \geq k$. The values of c and k must be fixed for the function f and must not depend on n.

Big-O

The importance of this measure can be seen in trying to decide whether an algorithm is adequate, but may just need a better implementation, or the algorithm will always be too slow on a big enough input. For instance, quicksort, which is O(n log n) on average, running on a small desktop computer can beat bubble sort, which is O(n^2), running on a supercomputer if there are a lot of numbers to sort. To sort 1,000,000 numbers, the quicksort takes 6,000,000 steps on average, while the bubble sort takes 1,000,000,000,000 steps!

Ω

A theoretical measure of the execution of an algorithm, usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation $f(n) = \Omega(g(n))$ means it is more than some constant multiple of $g(n)$. More formally, it means there are positive constants c and k, such that $0 \leq cg(n) \leq f(n)$

$$\boxed{\Omega(g(n)) = \omega(g(n)) \cup \Theta(g(n)).}$$

for all $n \geq k$. The values of c and k must be fixed for the function f and must not depend on n.

Some properties

Asymptotic	Transitivity	Reflexivity	Symmetry
Little-o	Yes	No	No
ω	Yes	No	No
Θ	Yes	Yes	Yes
Big-O	Yes	Yes	No
Ω	Yes	Yes	No

Some properties

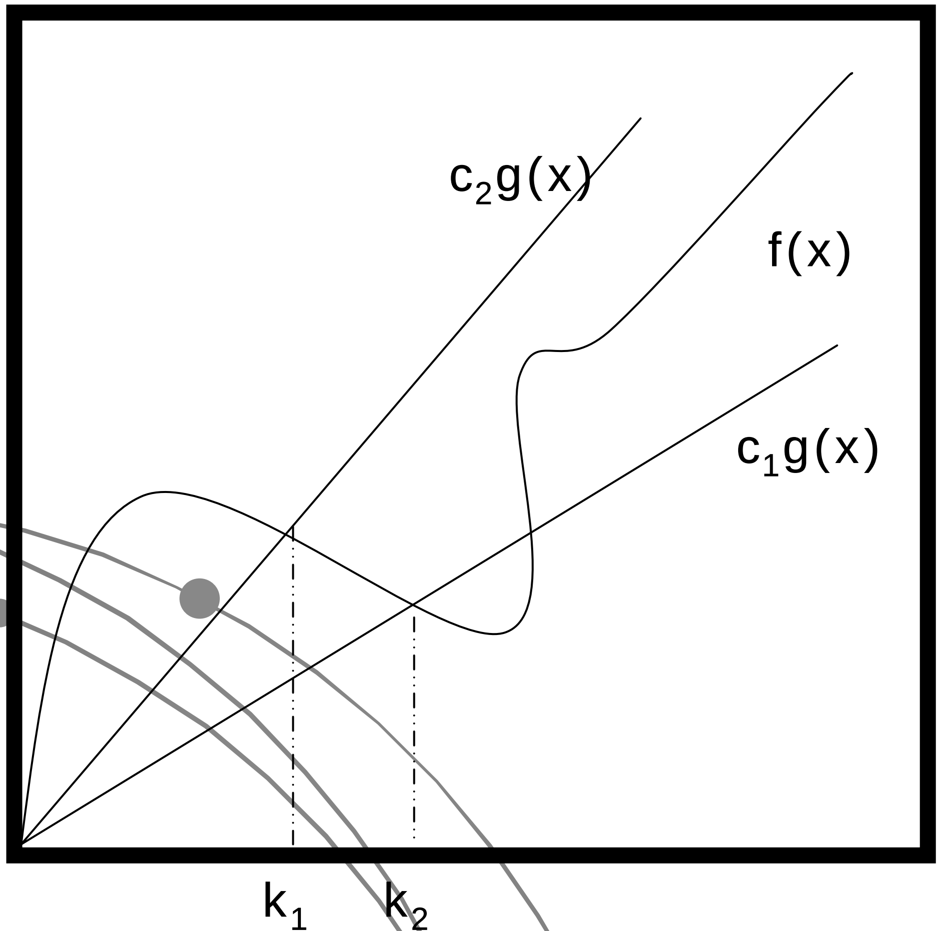
Asymptotic	Transitivity	Reflexivity	Symmetry
Little-o	Yes	No	No
ω	Yes	No	No
Θ	Yes	Yes	Yes
Big-O	Yes	Yes	No
Ω	Yes	Yes	No

Transpose Symmetry

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Asymptotic Notations



$f(x) = \Omega(g(x))$
For all $x \geq k_2$,
$f(x) \geq c_1g(x)$.

$f(x) = O(g(x))$
For all $x \geq k_1$,
$f(x) \leq c_2g(x)$.

$f(x) = \Theta(g(x))$
For all $x \geq k_1, k_2$,
$c_1g(x) \leq f(x) \leq c_2g(x)$.

Asymptotic Notations

SPECIAL ORDERS OF GROWTH

constant	: $\Theta(1)$
logarithmic	: $\Theta(\log n)$
polylogarithmic	: $\Theta(\log^c n)$, $c \geq 1$
sublinear	: $\Theta(n^a)$, $0 < a < 1$
linear	: $\Theta(n)$
quadratic	: $\Theta(n^2)$
polynomial	: $\Theta(n^c)$, $c \geq 1$
exponential	: $\Theta(c^n)$, $c > 1$

Exercises

- Show that
  - $\log n! = \Theta(n \log n)$,
  - $\log_a n = \Theta(\log_b n)$.

Analysis of algorithm

A detail unambiguous sequence of actions to perform to accomplish some task. Named after a Persian mathematician Abu Ja'far Mohammed ibn Mûsâ Al-Khawarizmi who wrote a book with arithmetic rules dating from about 825 A.D.

Al-Khawarizmi

CRITERIAs

- Correctness
- Amount of work done
- Amount of space used
- Simplicity
- Optimality

Maximum-Minimum search

```
001     begin
002     answer ← A[1]
003             for  I ← 2 step 1 until n do
004                     if A[I] > answer then answer = A[I] endif
005             enddo
006     return answer
007     end
```

Time used : $TA_1(n) = t_{002} + nt_{003} + (n\text{-}1)t_{004} + t_{006}$.

$$T_{A1}(n) = \Theta(t_{002}) + n\Theta(t_{003}) + (n\text{-}1)\,\Theta\,(t_{004}) + \Theta(t_{006})$$

$$T_{A1}(n) = \Theta\,(n)$$

Maximum-Minimum search

```
011    max-search (A[1..n])
012    begin
013        if n = 1 then return A[1] endif
014        answer ← max-search(A[1..n-1])
015        if A[n] > answer then return A[n] else return answer endif
016    end
```

Time used : $T_{A2}(n) = T_{A2}(n\text{-}1) + t_{013} + t_{014} + t_{015} = nt_{013} + (n\text{-}1)(t_{014}+t_{015})$.

$$\mathbf{T_{A2}(\boldsymbol{n}) = \boldsymbol{n}\Theta(\boldsymbol{t}_{013}) + (\boldsymbol{n}\text{-}1)(\Theta(\boldsymbol{t}_{014})+\Theta(\boldsymbol{t}_{015}))}$$

$$\mathbf{T_{A2}(\boldsymbol{n}) = \Theta(\boldsymbol{n})}$$

RUNNING TIME ANALYSIS

To simplify running time analysis,

- Count only “barometer” instructions
- Use asymptotic analysis

Basic instruction

To analyze total time used of an algorithm, we usually translate each operation into basic operation such that it has a constant time complexity, $\Theta(1)$.

Example: Which statements are basic?

STATEMENT	Y/N
if a < b then a $\leftarrow$ a+1 else a $\leftarrow$ a-1 endif.	YES
if b < c then e $\leftarrow$ c! endif	NO
d $\leftarrow$ c^n	NO
goto 005	YES
call max(A[1..n])	NO
return $\leftarrow$ subroutine(A,B)	NO

Sorting Algorithm

```
001   sort(A[1..n])
002   begin
003   for last ← n step –1 until 2 do
004         for  i ← 2 step 1 until last do
005               if A[i] < A[i-1] then buffer ← A[i]
                                         A[i] ← A[i-1]
                                         A[i-1] ← buffer endif
006         enddo
007   enddo
008   end
```

Operation	Times	Complexity
003	n	$\Theta(n)$
004	$\sum_{2 \le last \le n} last$	$\Theta(n(n+1)/2 - 1)$
005	$\sum_{2 \le last \le n} last-1$	$\Theta(n(n-1)/2)$
	Total	$\Theta(n^2)$

Sequencing

Give two operations, S_1 and S_2, such that S_2 will be processed after S_1 sequentially. Total time complexity depends on the complexity of each operation.

- **Upper bound of S_1S_2 = max(upper bound S_1, upper bound S_2)**
- **Lower bound of S_1S_2 = max(lower bound S_1, lower bound S_2)**

Sequencing

S_1	Time low/up	S_2	Time low/up	Time low/up	S_1S_2
$\Theta(n)$		$\Theta(n)$			
$\Theta(n)$		$O(n)$			
$\Theta(n)$		$O(n^2)$			
$\Theta(n^2)$		$\Theta(n)$			

Sequencing

S_1	Time low/up	S_2	Time low/up	Time low/up	S_1S_2
$\Theta(n)$	$\Omega(n)$ $O(n)$	$\Theta(n)$	$\Omega(n)$ $O(n)$	$\Omega(n)$ $O(n)$	$\Theta(n)$
$\Theta(n)$	$\Omega(n)$ $O(n)$	$O(n)$	unknown $O(n)$	$\Omega(n)$ $O(n)$	$\Theta(n)$
$\Theta(n)$	$\Omega(n)$ $O(n)$	$O(n^2)$	unknown $O(n^2)$	$\Omega(n)$ $O(n^2)$	$O(n^2)$
$\Theta(n^2)$	$\Omega(n^2)$ $O(n^2)$	$\Theta(n)$	$\Omega(n)$ $O(n)$	$\Omega(n^2)$ $O(n^2)$	$\Theta(n^2)$

If statement

Given three commands, S_1, S_2 and S_3.

Let t_1, t_2 and t_3 be time complexity of S_1, S_2 and S_3 respectively.

Consider a statement

if S_1 **then** S_2 **else** S_3 **endif**

Let T be time complexity of this statement, we have that:

- Lower bound of $T = t_1 + \min(t_2,t_3)$
- Upper bound of $T = t_1 + \max(t_2,t_3)$

If statement

Example:

Find the time complexity of **if** S_1 **then** S_2 **else** S_3 **endif** if $t_1 = \Theta(2^n)$, $t_2 = O(n)$ and $t_3 = O(n \log n)$.

$$\begin{aligned}
\text{Upper bound of } T &= \Omega(2^n) + \min(\text{ unknown, unknown })\\
&= \Omega(2^n).\\
\text{Lower bound of } T &= O(2^n) + \max(\, O(n), O(n \log n)\,)\\
&= O(2^n) + O(n \log n)\\
&= O(2^n).
\end{aligned}$$

Time complexity is $\Theta(2^n)$.

If statement

Exercise

Find the time complexity of

if $\Theta(n \log n)$ then $O(n^2)$ else $\Theta(\log n)$ endif

$O(n^2)$

if $O(n^3)$ then $O(n \log n)$ else $\Theta(2^n)$ endif

$O(2^n)$

if $O(n!)$ then $\Theta(n^2)$ else $\Theta(n)$ endif

$O(n!)$

if $\Theta(n)$ then $O(n \log n)$ else $\Theta(n \log n)$ endif

$O(n \log n)$

For-loop Statement

Time complexity can be computed by summary of time of each loop.

Let t_i be time used in the i^{th} iteration. Then total time used is equal to

$$\Sigma_{\text{all } i}\, t_i.$$

For-loop Statement

Example: Find time complexity of

```
for last ← n step –1 until 2 do
    for i ← 2 step 1 until last do
        if A[i] < A[i-1] then swap(A[i],A[i-1]) endif
    enddo
enddo
```

Time complexity

$$
\begin{aligned}
&= \Sigma_{2\le last\le n}\, \Sigma_{2\le i\le last+1}\, \Theta(1) \\
&= \Sigma_{2\le last\le n}\, \Theta(last) \\
&= \Theta(\Sigma_{2\le last\le n}\, last) \\
&= \Theta(n(n+1)/2 - 1) \\
&= \Theta(n^2).
\end{aligned}
$$

For-loop Statement

Example: Find time complexity of

```
for i ← 1 step 1 until m do
    for j ← m step –1 until 1 do
        for k ← 1 step 1 until j do
            sum ← sum + i + j + k
        enddo
    enddo
enddo
```

Time complexity

$$
\begin{aligned}
&= \Sigma_{1\le i\le m} (\Sigma_{1\le j\le m} (\Sigma_{1\le k\le j} (\Theta(1))) \\
&= \Sigma_{1\le i\le m} (\Sigma_{1\le j\le m} \Theta(\Sigma_{1\le k\le j} k)) \\
&= \Sigma_{1\le i\le m} (\Sigma_{1\le j\le m} \Theta(j)) \\
&= \Sigma_{1\le i\le m} (\Theta(m^2)) \\
&= \Theta(m^3)
\end{aligned}
$$

While-loop Statement

By the same way, we have to count the number of iterations.

Example: Find the time complexity of

```
while (n > 0) do
    n ← ⌊n/2⌋
enddo
```

Consider the value of n in each loop. For the i-th iteration, the value of n is equal to $n/2^i$*. Since the number of iterations is equal to* $\log_2 n$,

$$\text{time complexity} = \Theta(\log_2 n).$$

While-loop Statement

Example: Fine the complexity of

```
for i ← 1 step 1 until n-1 do
        j ← i.
        while j > 0 and A[j] < A[j-1] do
                sum ← sum + A[j]
                j ← j - 1
        enddo
enddo
```

Time complexity = $\Theta(n) + \Sigma_{1 \le i \le n-1} (O(\Sigma_{1 \le j \le i} 1)) = O(n^2)$.

While-loop Statement

Definition: An algorithm to compute the *greatest common divisor* of two integers. It is Euclid(a,b){if (b=0) then return a; else return Euclid(b, a mod b);}.

Algorithm Euclid's GCD

```
GCD(a,b)
begin
while ( a > o ) do
    q ← a
    a ← b mod a
    b ← q
enddo
return b
end
```

b > a

Recursive calls

It is important to see that the size of problem should be decreased for each iteration of recursive called. We can count all instructions by

1. Set run time complexity = T(n) where n is the size of problem.
2. Time complexity of recursive call statement = T(m) where m is the input size and $m < n$.
3. Let T(n) = T(m) + time complexity of other instructions.
4. Find T(n) in term of asymptotic complexity function.

Recursive calls

Algorithm Selection Sort

```
SelectionSort(A{1..n])
begin
if ( n ≤ 1 ) then return endif
maxindex ← Max(A[1..n])
    buffer ← A[n]
    A[n] ← A[maxindex]
    A[maxindex] ← buffer
SelectionSort(A[1..n-1])
end
```

Time complexity $T(n)$

$$
\begin{aligned}
&= T(n-1) + \Theta(n) \\
&= T(n-2) + \Theta(n-1) + \Theta(n) \\
&= \dots \\
&= \Sigma_{1 \le i \le n}\, \Theta(i) = \Theta(n^2).
\end{aligned}
$$

Recursive calls

Algorithm Binary Search

```
BinarySearch(A[1..n],x)
begin
if ( A[1] > A[n] ) then return –1 endif
mid ← (A[1] + A[n])/2
if ( x = A[mid] ) then return mid endif
if ( x < A[mid] ) then return BinarySearch(A[1..mid-1],x)
     else return BinarySearch(A[mid+1..n],x) endif
end
```

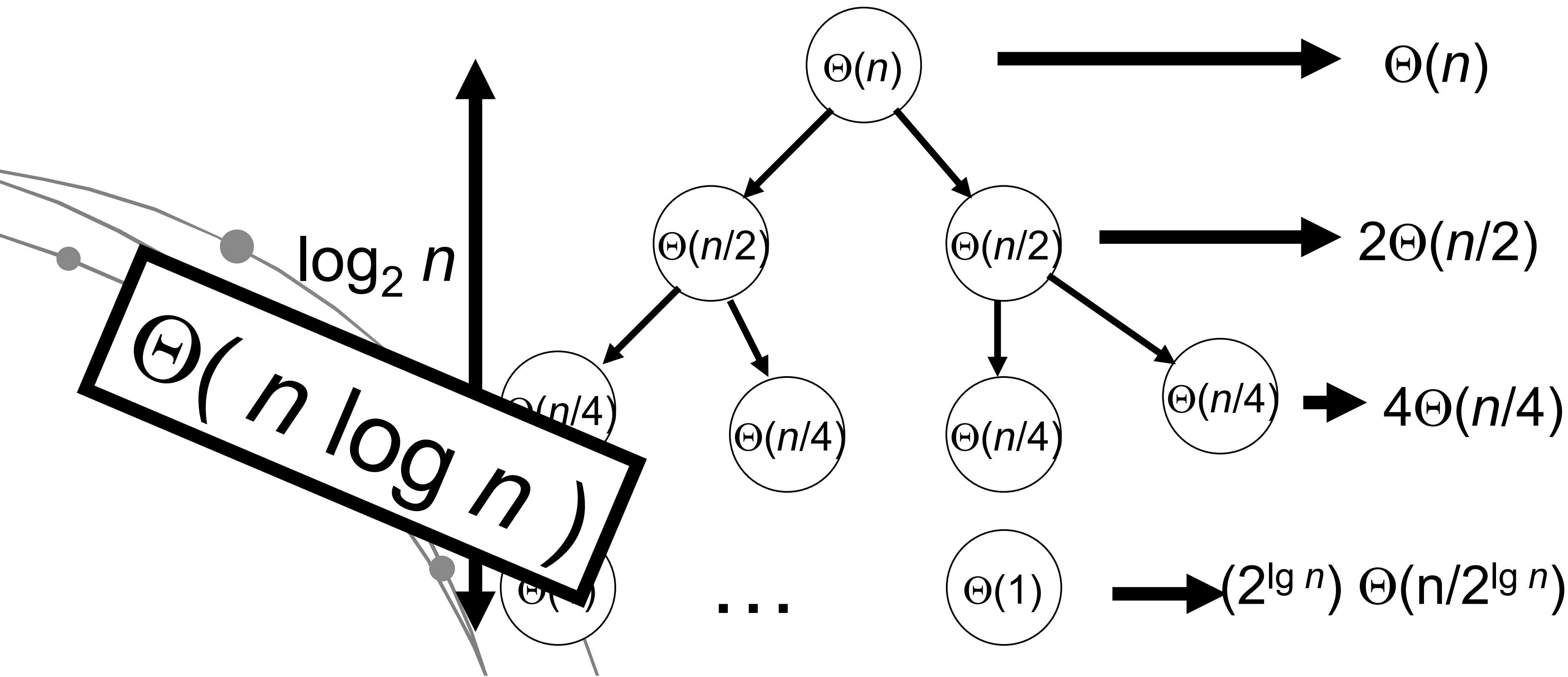
Time complexity $T(n)$

$$
\begin{aligned}
T(n) &\le T(n/2) + \Theta(1) \\
&\le T(n/2^2) + \Theta(1) + \Theta(1) \\
&\le T(n/2^k) + \Sigma_{1\le i\le k}\, \Theta(i) \\
&\le \Theta(1) + \Sigma_{1\le i\le \log n}\, \Theta(1) \\
&\le \Theta(\log_2 n) \\
&= O(\log_2 n).
\end{aligned}
$$

Recursive calls

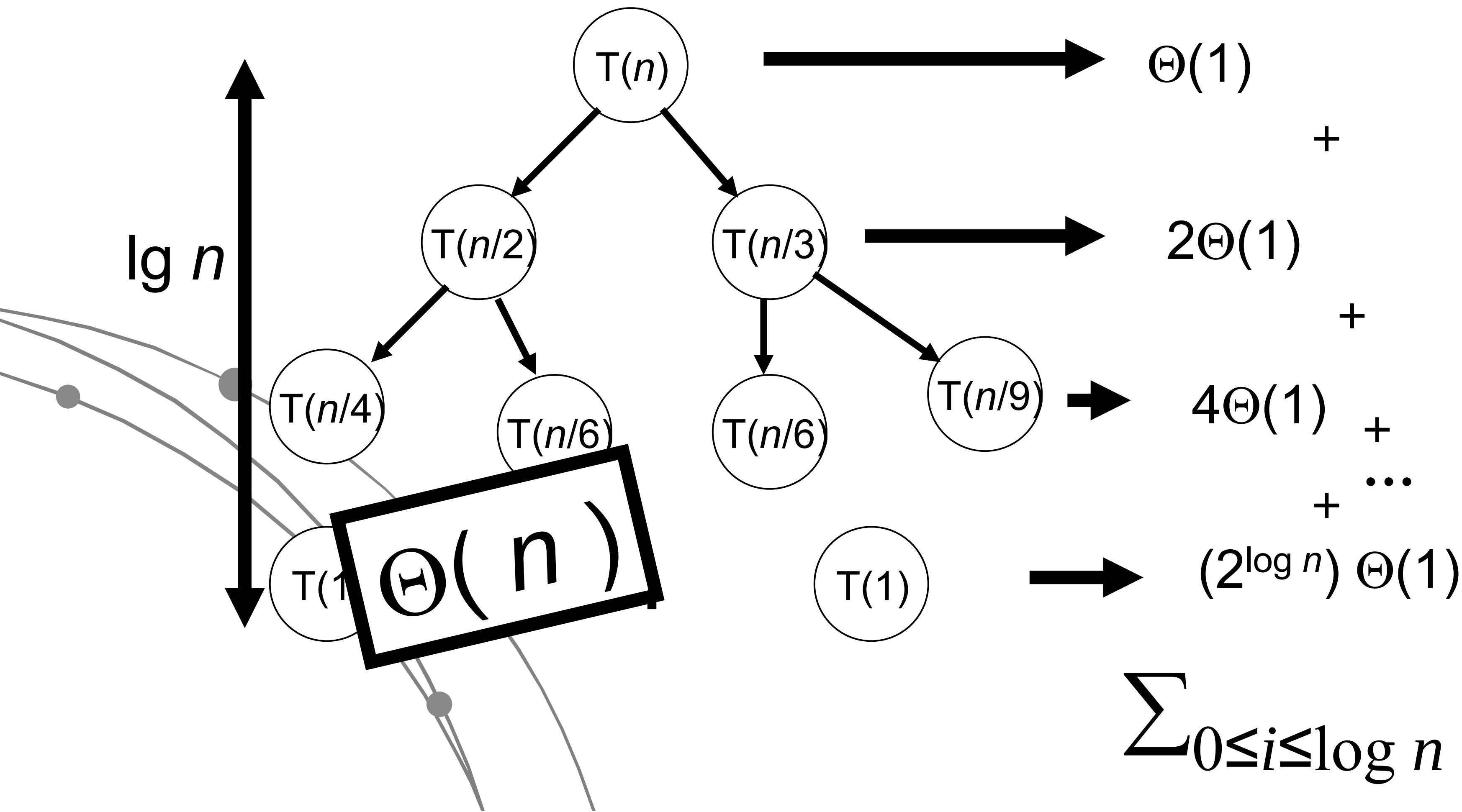
The recursive call in a given algorithm can be represented by using recursive tree. For example, the run time complexity of a given algorithm is

$T(n) = 2T(n/2) + \Theta(n)$, for any $n>1$ and $T(1) = \Theta(1)$.



Recursive calls

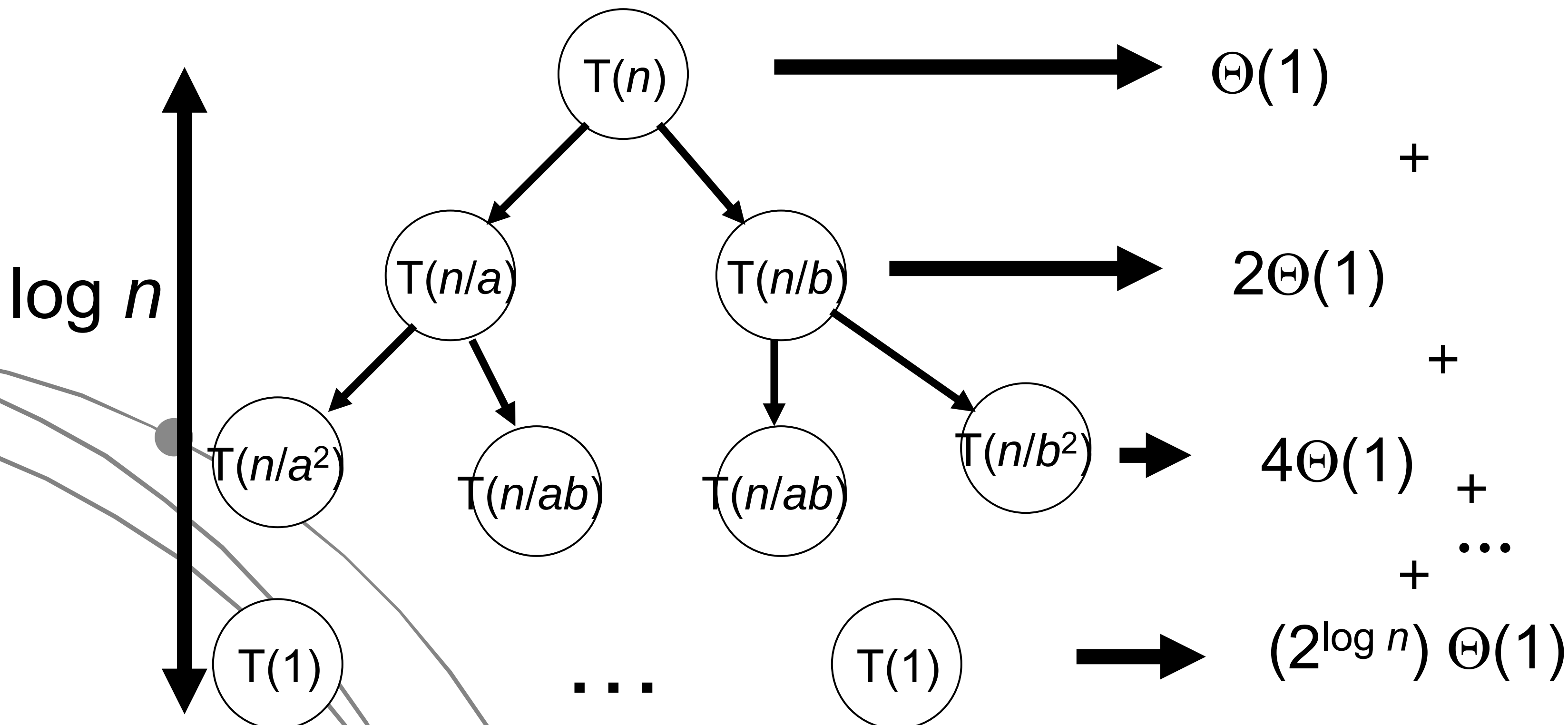
T(n) = T(n/2) + T(n/3), for any n>1 and T(1) = Θ(1).



$$\sum_{0\leq i\leq \log n} 2^i$$

Recursive calls

$T(n) = T(n/a) + T(n/b)$, for any $n>1$ and $T(1) = \Theta(1)$.

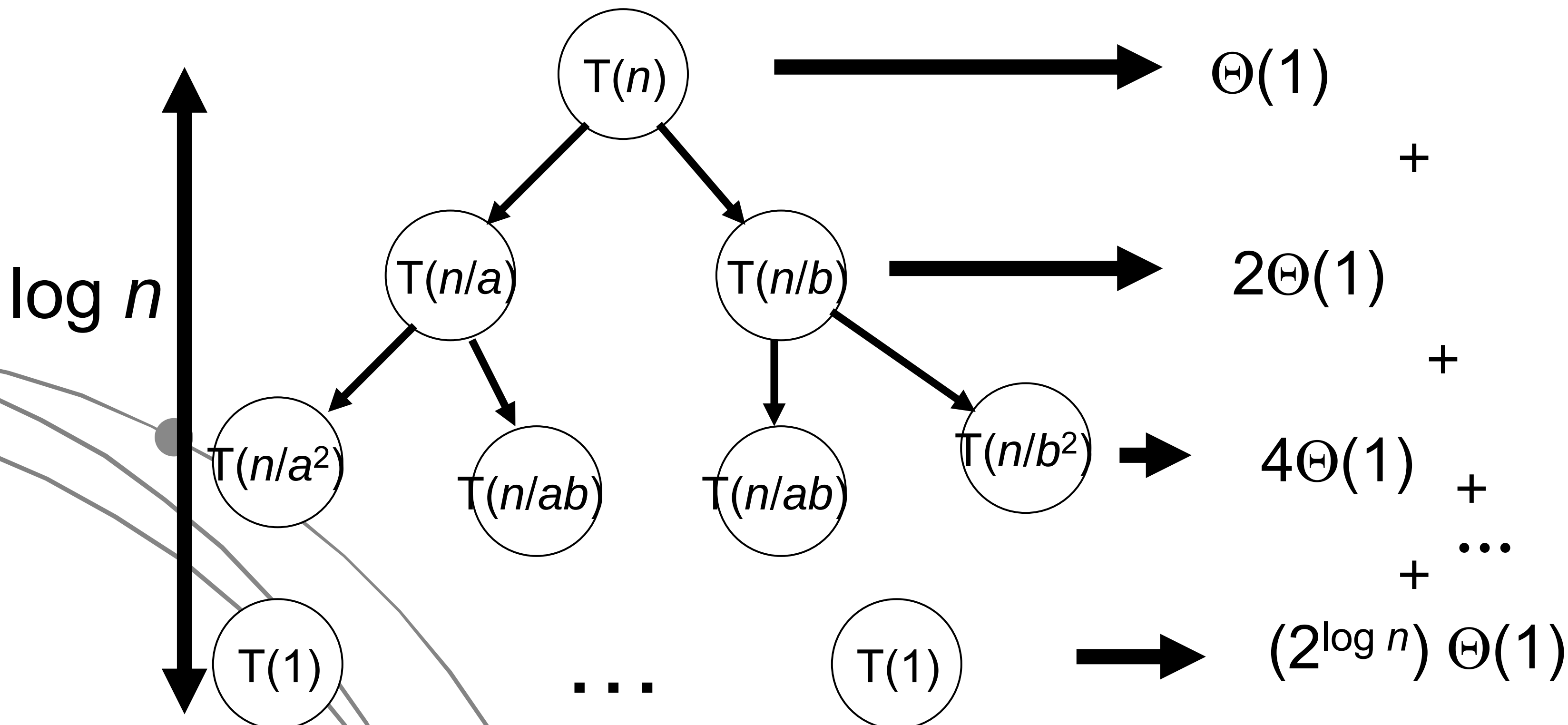


$\boxed{a>1 \text{ and } b>1}$

$$\sum_{0\leq i\leq \log n} 2^i$$

Recursive calls

$$T(n) = T(n/a) + T(n/b) + \Theta(n), \text{ for any } n>1$$
$$\text{and } T(1) = \Theta(1).$$

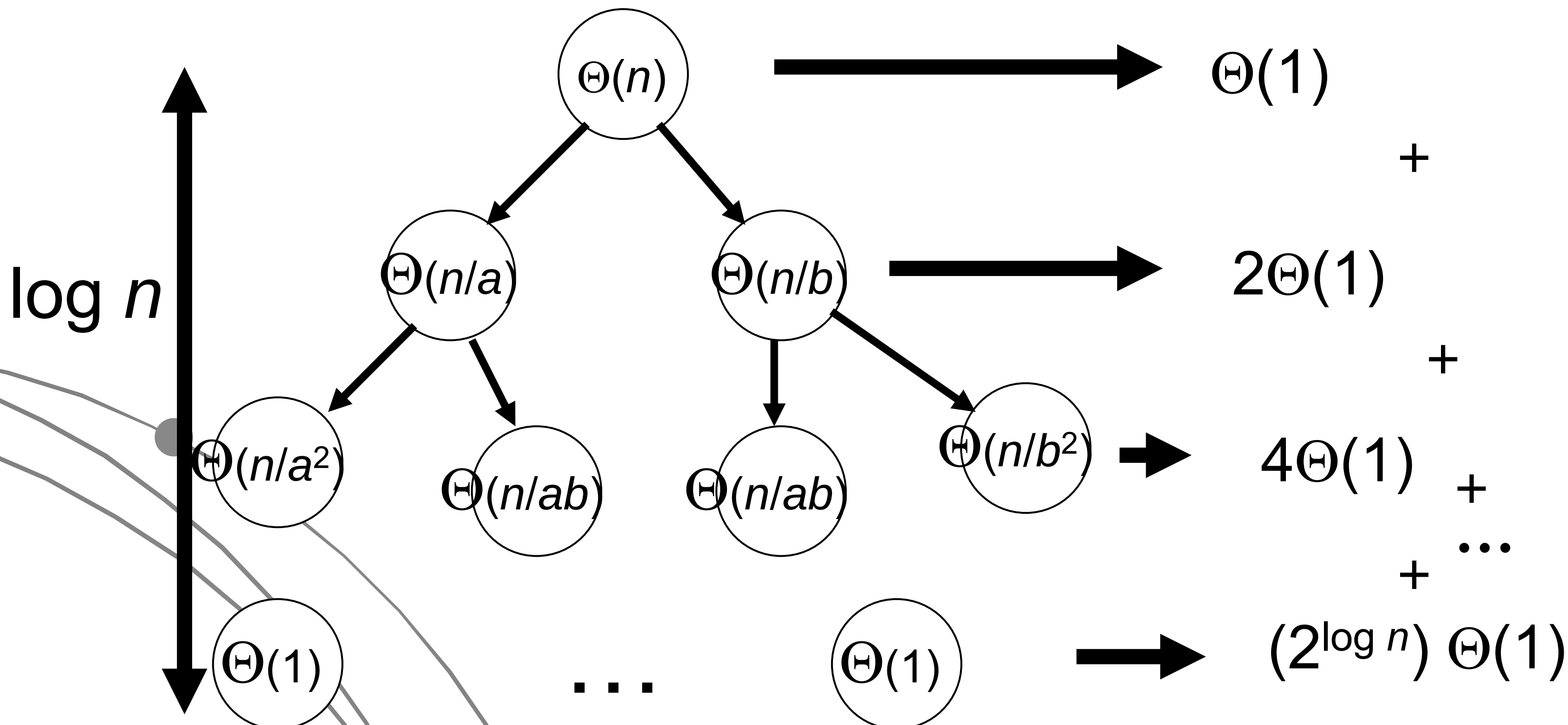


$\boxed{a>1 \text{ and } b>1}$

$$\sum_{0\le i\le \log n} 2^i$$

Recursive calls

$T(n) = T(n/a) + T(n/b) + \Theta(n)$, for any $n>1$
and $T(1) = \Theta(1)$.

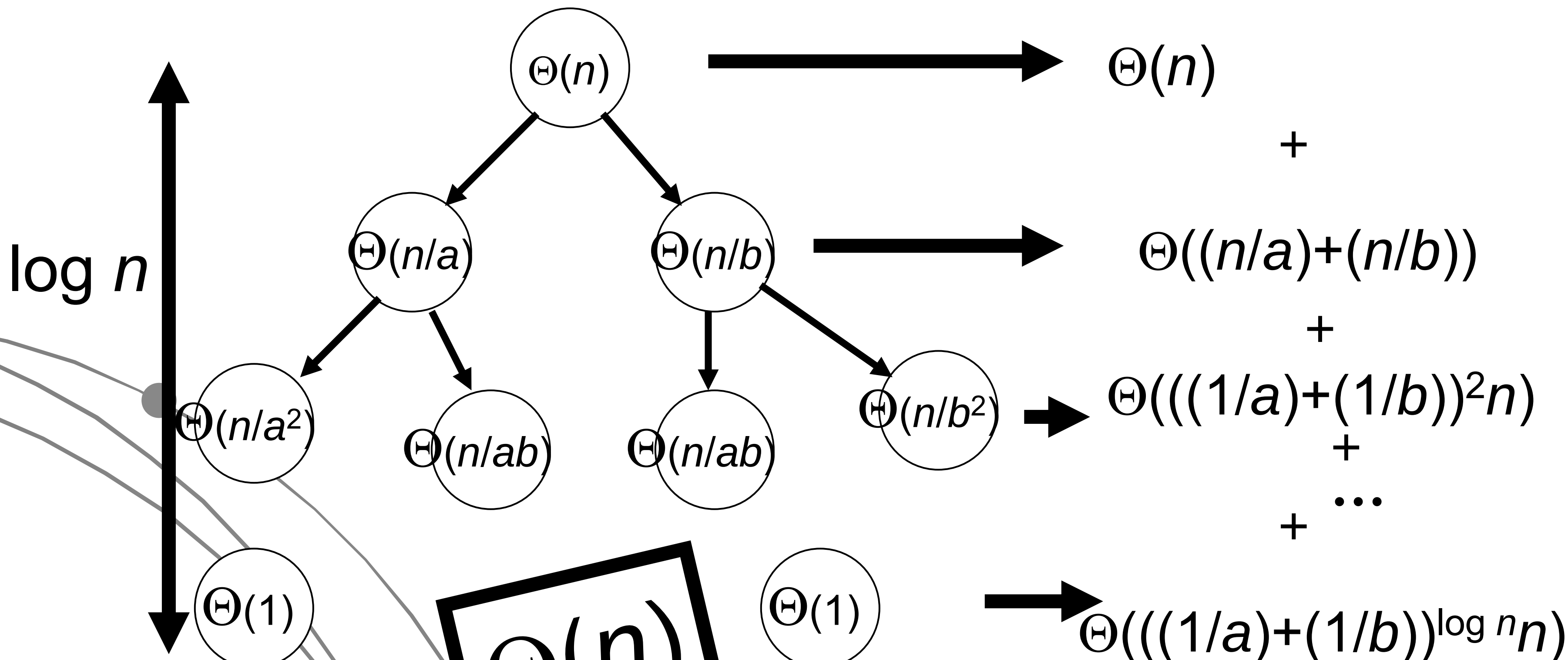


$\boxed{a>1 \text{ and } b>1}$

$$\sum_{0\leq i\leq \log n} 2^i$$

Recursive calls

$T(n) = T(n/a) + T(n/b) + \Theta(n)$, for any $n>1$
and $T(1) = \Theta(1)$.



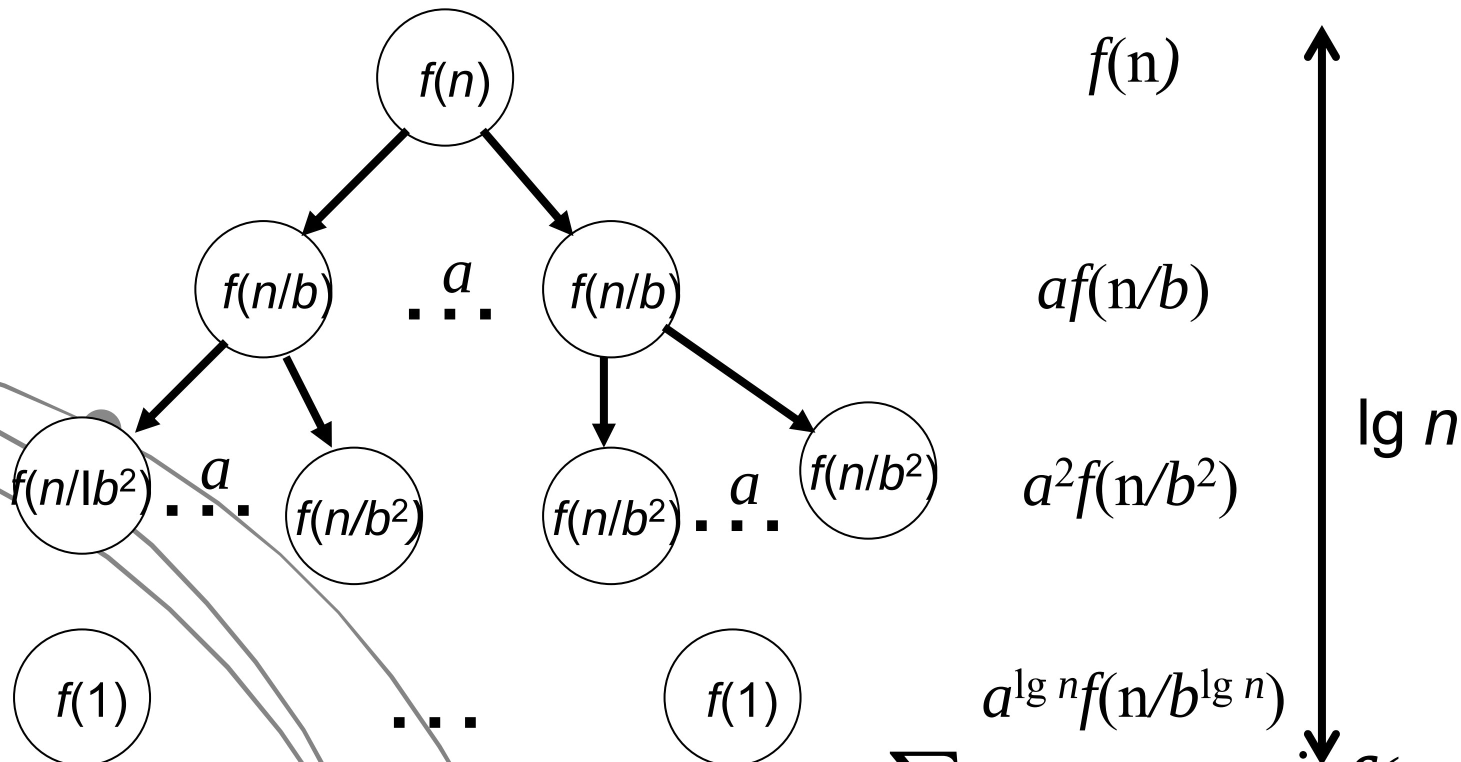
$\boxed{a>1 \text{ and } b>1}$

$\boxed{\Theta(n)}$

$$\sum_{0 \le i \le \log n} ((1/a)+(1/b))^i n$$

Master method

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b \geq 1$



$$\sum_{1 \leq i \leq \lg n} a^i f(n/b^i)$$

Master method

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b \geq 1$

This complexity depends on three variables, a, b and $f(n)$.

$f(n)$	$T(n)$
$O(n^{(\log_b a)-\varepsilon})$, for fixed $\varepsilon > 0$	$O(n^{(\log_b a)})$

This means that the growth rate of $f(n)$ is less than $O(n^{(\log_b a)})$. Then total time complexity is $O(n^{(\log_b a)})$.

Master method

$$\mathbf{T(\mathit{n}) = \mathit{a}T(\mathit{n}/\mathit{b}) + \mathit{f}(\mathit{n})\ where\ \mathit{a} \geq 1\ and\ \mathit{b} \geq 1}$$

This complexity depends on three variables, a, b and $f(n)$.

$f(n)$	$T(n)$
$O(n^{(\log_b a)-\varepsilon})$, for fixed $\varepsilon > 0$	$O(n^{(\log_b a)})$
$\Theta(n^{(\log_b a)})$	$\Theta(n^{(\log_b a)} \log n)$

It is clear that total time complexity is equal to $\Theta(n^{(\log_b a)} \log n)$.

Master method

T(*n*) = *a*T(*n*/*b*) + *f*(*n*) where *a* ≥ 1 and *b* ≥ 1

This complexity depends on three variables, *a*, *b* and *f*(*n*).

This means that the growth rate of $f(n)$ is more than $\Omega(n^{(\log_b a)})$, and the condition $af(n/b) \le cf(n)$ for $c < 1$ and $n > n_0$ means that the growth rate of $f(n)$ decreases where n increases. Then total growth rate is $\Theta(f(n))$.

$\Omega(n^{(\log_b a)+\varepsilon})$, for fixed $\varepsilon > 0$ and $af(n/b) \le cf(n)$ for $c < 1$ and $n > n_0$	$\Theta(f(n))$

Master method

Example: Find the solution of

- $T(n) = 16T(n/4) + n$
- $T(n) = 27T(n/3) + n^3$
- $T(n) = 3T(n/4) + n \log n$.

$T(n) = 16T(n/4) + n$
Since $\log_4 16 - 0.1 > 1$,
we can conclude that time complexity = $O(n^2)$.

Master method

Example: Find the solution of

- $T(n) = 16T(n/4) + n$
- $T(n) = 27T(n/3) + n^3$
- $T(n) = 3T(n/4) + n \log n$.

$T(n) = 27T(n/3) + n^3$
Since $\log_3 27 = 3$, it is clear that
time complexity = $\Theta(n^3 \log n)$.

Master method

Example: Find the solution of

- $T(n) = 16T(n/4) + n$
- $T(n) = 27T(n/3) + n^3$
- $T(n) = 3T(n/4) + n \log n$.

$T(n) = 3T(n/4) + n \log n$.
Since $\log_4 3 < 0.8$, and $f(n) = n \log n = \Omega(n^{0.8+0.2})$.
We also have that $3f(n/4) \le (3n/4)\log n$.
Then time complexity = $\Theta(n \log n)$.

Amortized analysis

Three methods

Aggregate method
Accounting method
Potential method