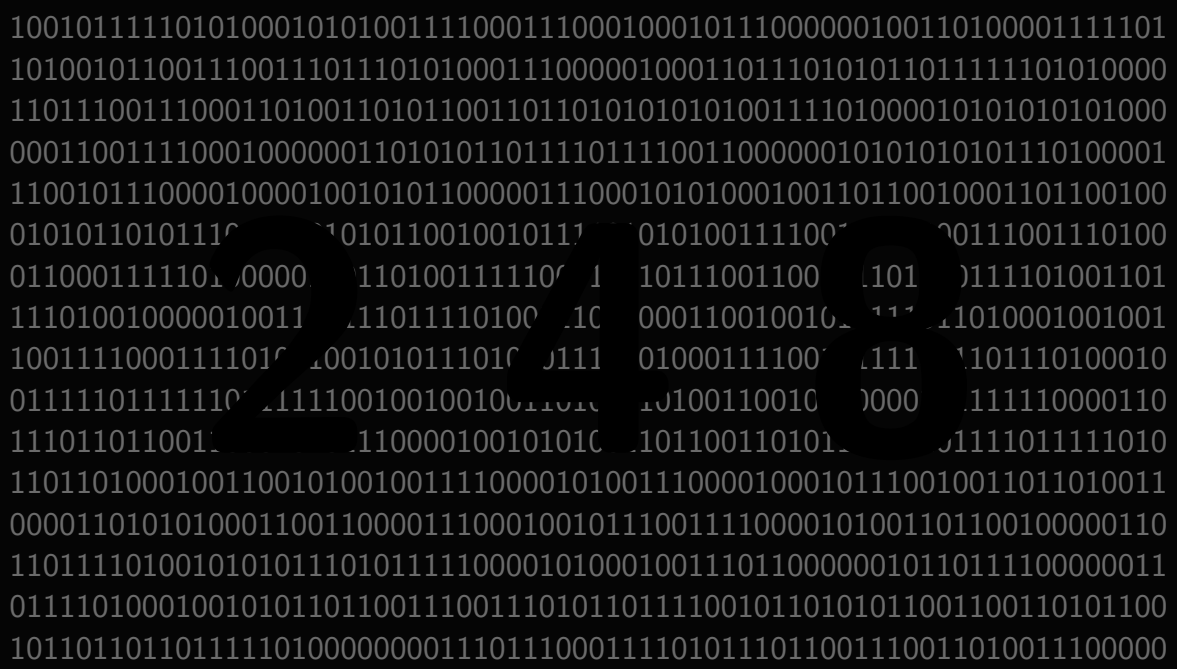
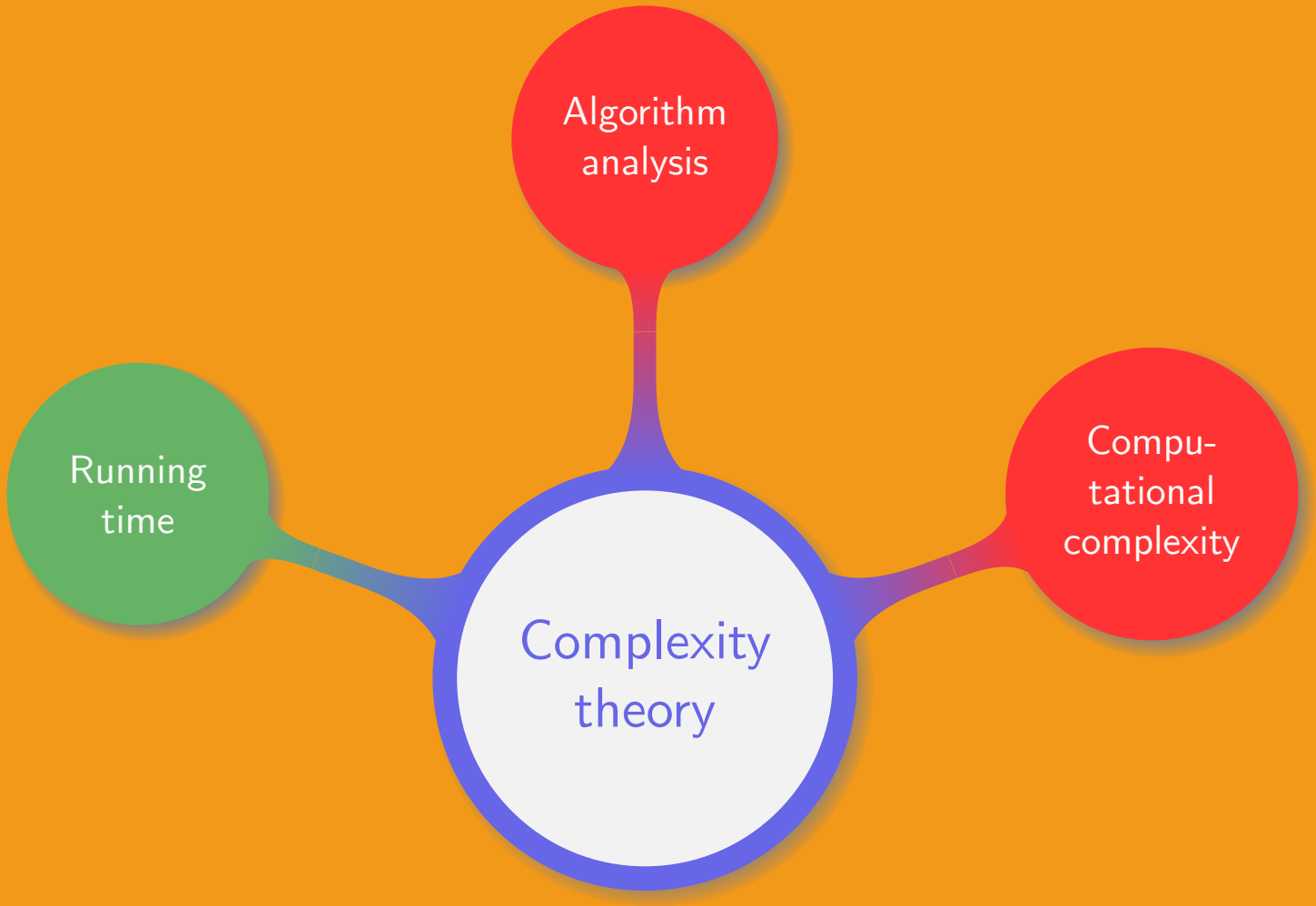


Introduction to Algorithms

2. Complexity theory

Manuel – Fall 2020

Chapter organisation



Preview of the RAM model

Random Access Machine (RAM) model:

- Each simple operation ($+, -, \times, /$, if,...) takes one step
- Loops are composed of several simple steps, which are repeated a certain number of times
- Each memory access accounts for one step

The runtime of an algorithm is given by the total number of steps necessary to complete it.

The RAM model allows the study of algorithms independently of their implementation or running environment.

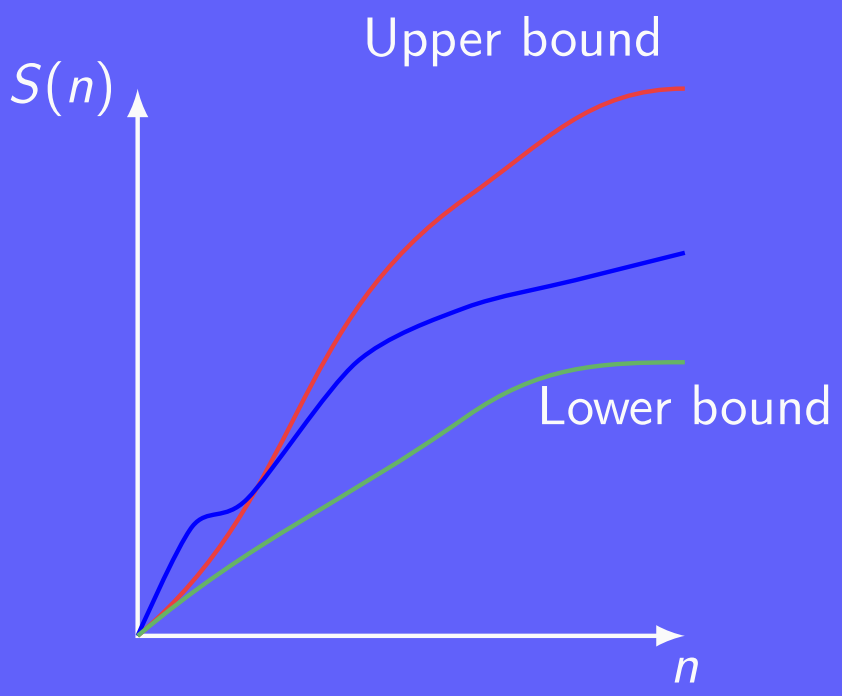
Time complexity

Given a problem with input size n:

- Worst case complexity: maximum number of steps to complete an instance of the algorithm
- Best case complexity: minimum number of steps to complete an instance of the algorithm
- Average case complexity: average number of steps, over all possible instances

The complexity of an algorithm is defined by a numerical function.

Approximation

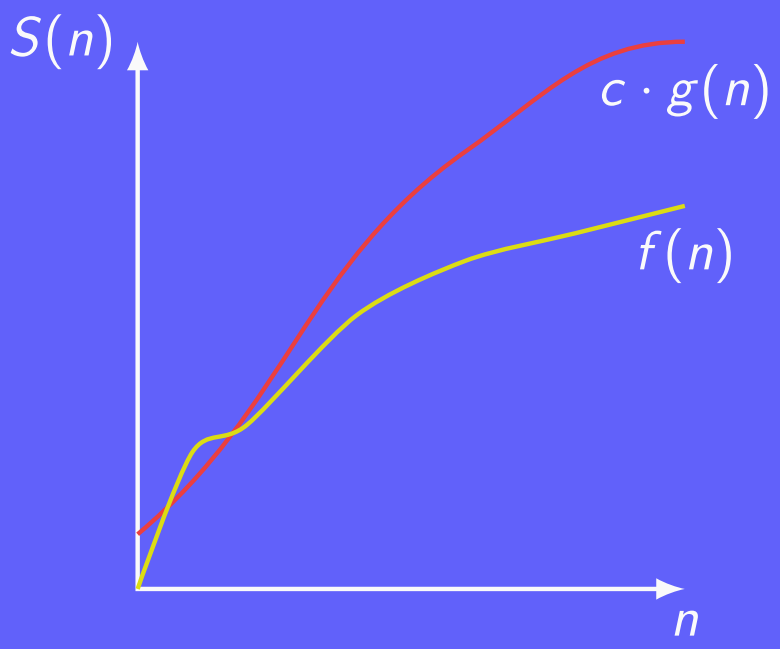


Limitations of the RAM:

- Complexity can display many bumps, e.g. algorithms performing better on input of size a power of 2
- Counting the exact number of operations is too complex, e.g. implementation choices impact the number of RAM instructions

We need notations that simplify the analysis by ignoring the details irrelevant to the comparison of the algorithms.

The $\mathcal{O}$ notation

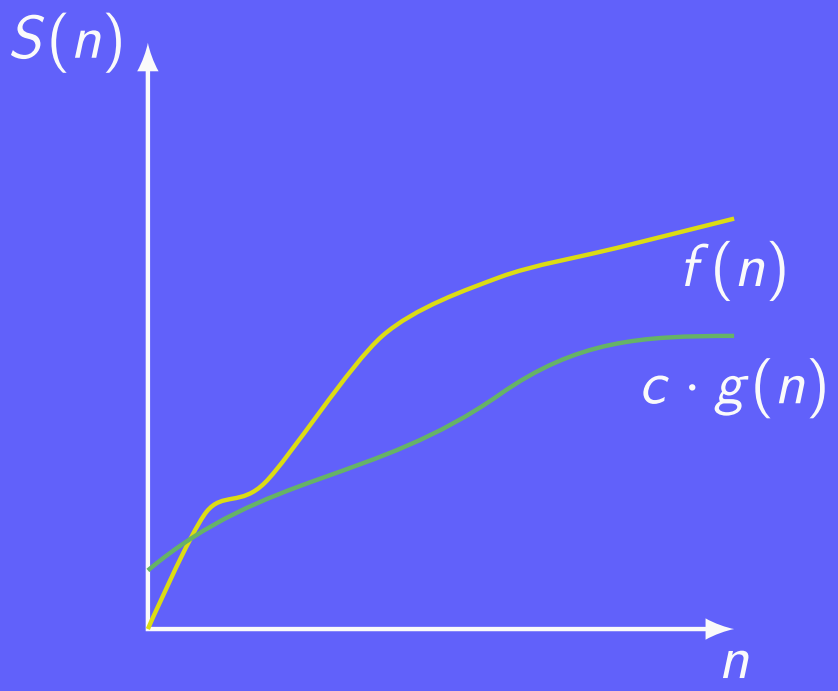


Approximating complexity:

- $c \cdot g(n)$ is an upper bound for $f(n)$
- There exists two constants c and n_0 such that for all n larger than n_0, $f(n) \leq c \cdot g(n)$
- $f(n) = \mathcal{O}(g(n))$

An algorithm in $\mathcal{O}(f)$ has time complexity upper-bounded by f

The Ω notation

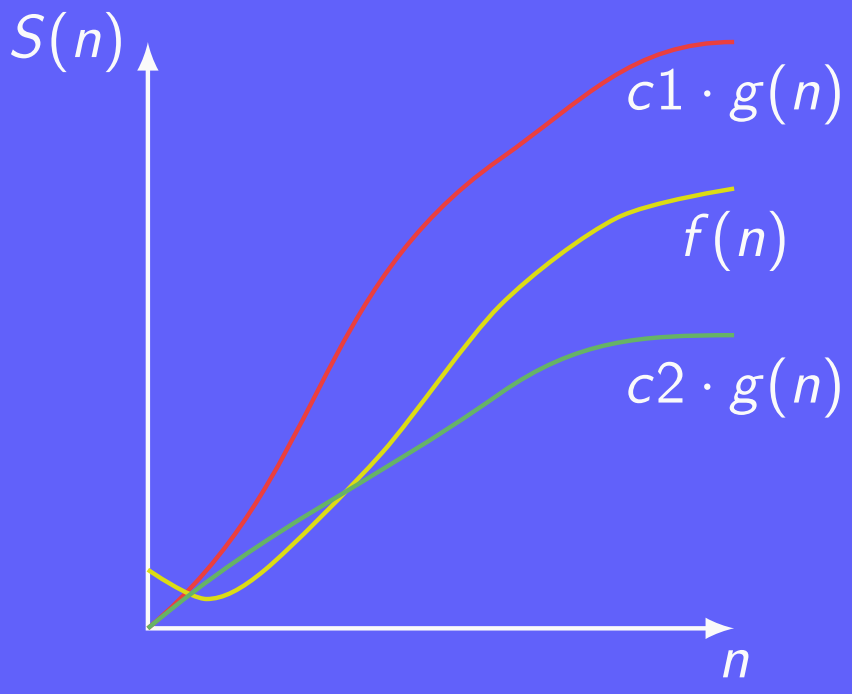


Approximating complexity:

- $c \cdot g(n)$ is a lower bound for $f(n)$
- There exists two constants c and n_0 such that for all n larger than n_0, $f(n) \geq c \cdot g(n)$
- $f(n) = \Omega(g(n))$

An algorithm in $\Omega(f)$ has time complexity lower-bounded by f

The Θ notation



Approximating complexity:

- $c_1 \cdot g(n)$ is an upper bound for $f(n)$ while $c_2 \cdot g(n)$ is a lower bound for $f(n)$
- There exists three constants c_1, c_2, and n_0 such that for all n larger than n_0, $c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$
- $f(n) = \Theta(g(n))$

An algorithm in $\Theta(f)$ has time complexity roughly similar to f

Example

Exercise. Prove the following equalities:

- $2^{n+1} = \Theta(2^n)$?

Example

Exercise. Prove the following equalities:

- $2^{n+1} = \Theta(2^n)$?
  It suffices to prove that $2^{n+1} = \mathcal{O}(2^n)$ and $2^{n+1} = \Omega(2^n)$.
  First note that $2^{n+1} = 2 \cdot 2^n$, such that $2^{n+1} \leq c \cdot 2^n$, for any $c \geq 2$.
  Then observe that for all $n \geq 2$, $2^{n+1} \geq c \cdot 2^n$, for any $0 < c \leq 2$.
- $(x+y)^2 = \mathcal{O}(x^2 + y^2)$?

Example

Exercise. Prove the following equalities:

- $2^{n+1} = \Theta(2^n)$?
  It suffices to prove that $2^{n+1} = \mathcal{O}(2^n)$ and $2^{n+1} = \Omega(2^n)$.
  First note that $2^{n+1} = 2 \cdot 2^n$, such that $2^{n+1} \leq c \cdot 2^n$, for any $c \geq 2$.
  Then observe that for all $n \geq 2$, $2^{n+1} \geq c \cdot 2^n$, for any $0 < c \leq 2$.
- $(x+y)^2 = \mathcal{O}(x^2 + y^2)$?
  We need to find some c such that $(x+y)^2 \leq c(x^2+y^2)$. Without loss of generality we can assume $x \geq y$. Then $2xy \leq 2x^2 \leq 2(x^2+y^2)$ and as $(x+y)^2 = x^2 + 2xy + y^2$, we conclude that $(x+y)^2 \leq 3(x^2+y^2)$.

Dominance relations

Common complexity functions

- Constant: $\mathcal{O}(1)$
- Logarithmic: $\mathcal{O}(\log n)$
- Linear: $\mathcal{O}(n)$
- Superlinear: $\mathcal{O}(n \log n)$
- Quadratic: $\mathcal{O}(n^2)$
- Cubic: $\mathcal{O}(n^3)$
- Exponential: $\mathcal{O}(c^n)$
- Factorial: $\mathcal{O}(n!)$

Dominance relations

Common complexity functions

- Constant: $\mathcal{O}(1)$
- Logarithmic: $\mathcal{O}(\log n)$
- Linear: $\mathcal{O}(n)$
- Superlinear: $\mathcal{O}(n \log n)$
- Quadratic: $\mathcal{O}(n^2)$
- Cubic: $\mathcal{O}(n^3)$
- Exponential: $\mathcal{O}(c^n)$
- Factorial: $\mathcal{O}(n!)$

More complexity functions

- Inverse Ackerman's function: $\mathcal{O}(\alpha(n))$
- Log Log: $\mathcal{O}(\log \log n)$
- $\mathcal{O}(\log n / \log \log n)$
- $\mathcal{O}(\sqrt{n})$
- $\mathcal{O}(n^{c+\varepsilon})$

Dominance relations

Common complexity functions

- Constant: $\mathcal{O}(1)$
- Logarithmic: $\mathcal{O}(\log n)$
- Linear: $\mathcal{O}(n)$
- Superlinear: $\mathcal{O}(n \log n)$
- Quadratic: $\mathcal{O}(n^2)$
- Cubic: $\mathcal{O}(n^3)$
- Exponential: $\mathcal{O}(c^n)$
- Factorial: $\mathcal{O}(n!)$

More complexity functions

- Inverse Ackerman's function: $\mathcal{O}(\alpha(n))$
- Log Log: $\mathcal{O}(\log \log n)$
- $\mathcal{O}(\log n / \log \log n)$
- $\mathcal{O}(\sqrt{n})$
- $\mathcal{O}(n^{c+\varepsilon})$

$$n! \gg 2^n \gg n^{c+\varepsilon} \gg n^3 \gg n^{2+\varepsilon} \gg n^2 \gg n \log n \gg n^{1+\varepsilon} \gg n$$

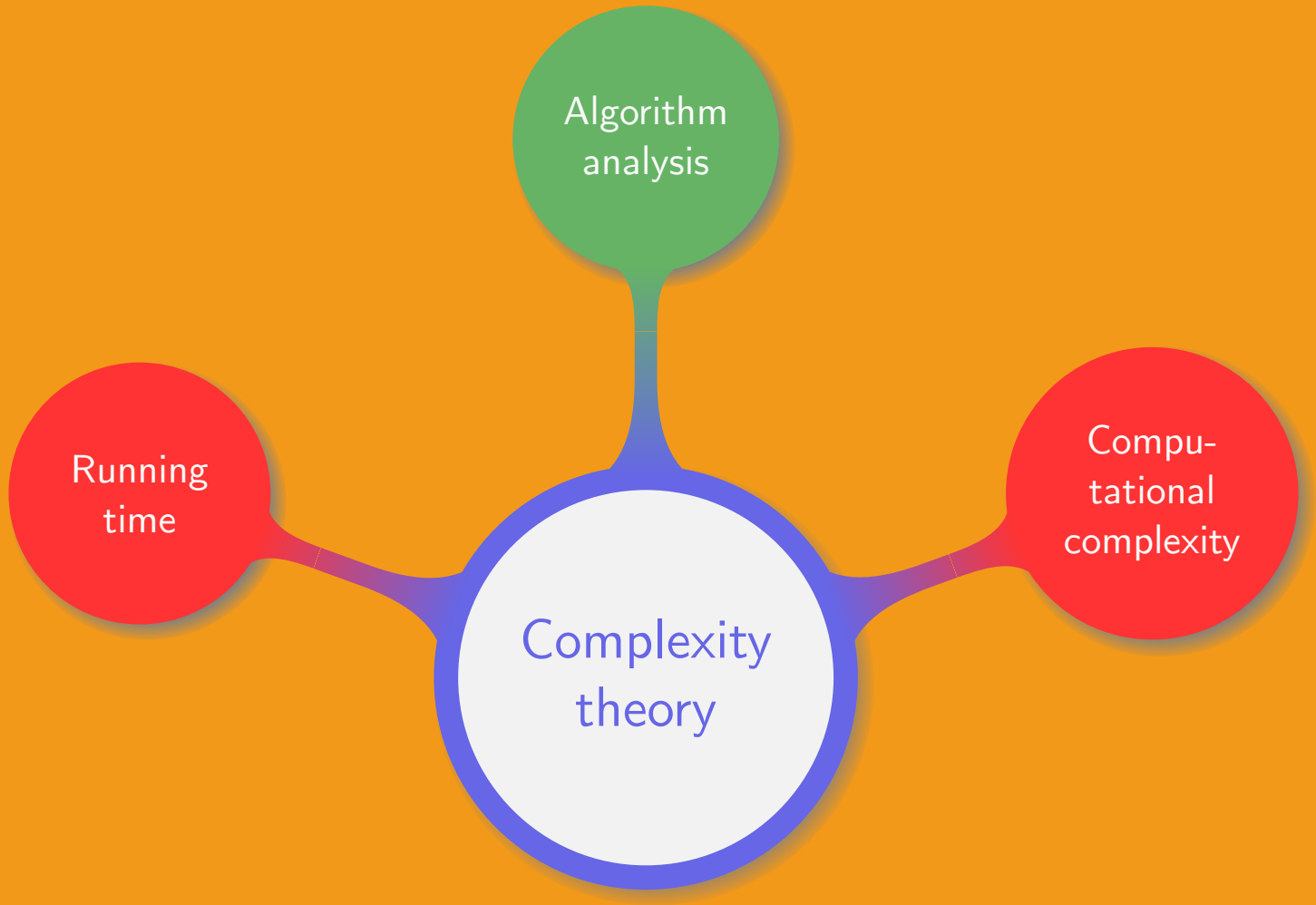
$$\gg \sqrt{n} \gg \log n \gg \frac{\log n}{\log \log n} \gg \log \log n \gg \alpha(n) \gg 1$$

Growth rate

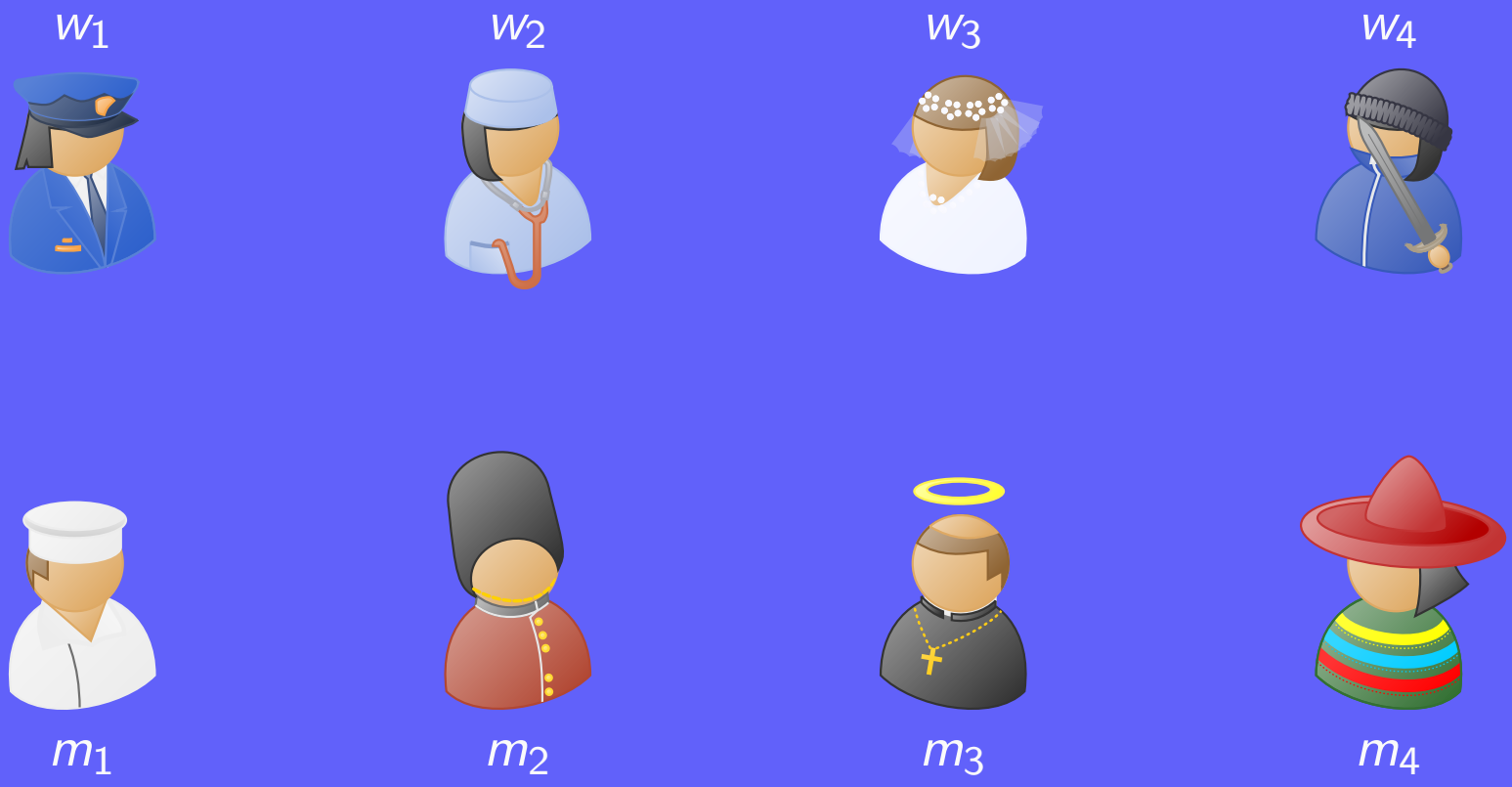
Assuming 10^9 basic operations a second, the time spent depending on the input size (in bits or number of input) and the time complexity is given as follows.

Input size n	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$< 0.01\mu s$	$0.01\mu s$	$0.03\mu s$	$0.1\mu s$	$1\mu s$	$1\mu s$	3.6ms
20	$< 0.01\mu s$	$0.02\mu s$	$0.09\mu s$	$0.4\mu s$	$0.8\mu s$	1ms	77y
30	$< 0.01\mu s$	$0.03\mu s$	$0.1\mu s$	$0.9\mu s$	$2.7\mu s$	1s	10^{16}y
40	$< 0.01\mu s$	$0.04\mu s$	$0.2\mu s$	$1.6\mu s$	$6.4\mu s$	18min	
50	$< 0.01\mu s$	$0.05\mu s$	$0.3\mu s$	$2.5\mu s$	$12.5\mu s$	13d	
100	$< 0.01\mu s$	$0.1\mu s$	$0.6\mu s$	$10\mu s$	1ms	10^{13}y	
1000	$0.01\mu s$	$1\mu s$	$10\mu s$	1ms	1s		
10000	$0.01\mu s$	$10\mu s$	$130\mu s$	100ms	16min		
100000	$0.02\mu s$	$100\mu s$	1.7ms	10s	11.5d		
1000000	$0.02\mu s$	1ms	20ms	17min	32y		
10000000	$0.02\mu s$	10ms	200ms	1.2d	32000y		
100000000	$0.03\mu s$	100ms	2.6s	116d			
1000000000	$0.03\mu s$	1s	30s	32y			

Chapter organisation



Stable marriage problem



Stable marriage problem

$m_3 > m_2 > m_1 > m_4$ $m_4 > m_1 > m_2 > m_3$ $m_3 > m_4 > m_1 > m_2$ $m_1 > m_2 > m_3 > m_4$

w_1 w_2 w_3 w_4







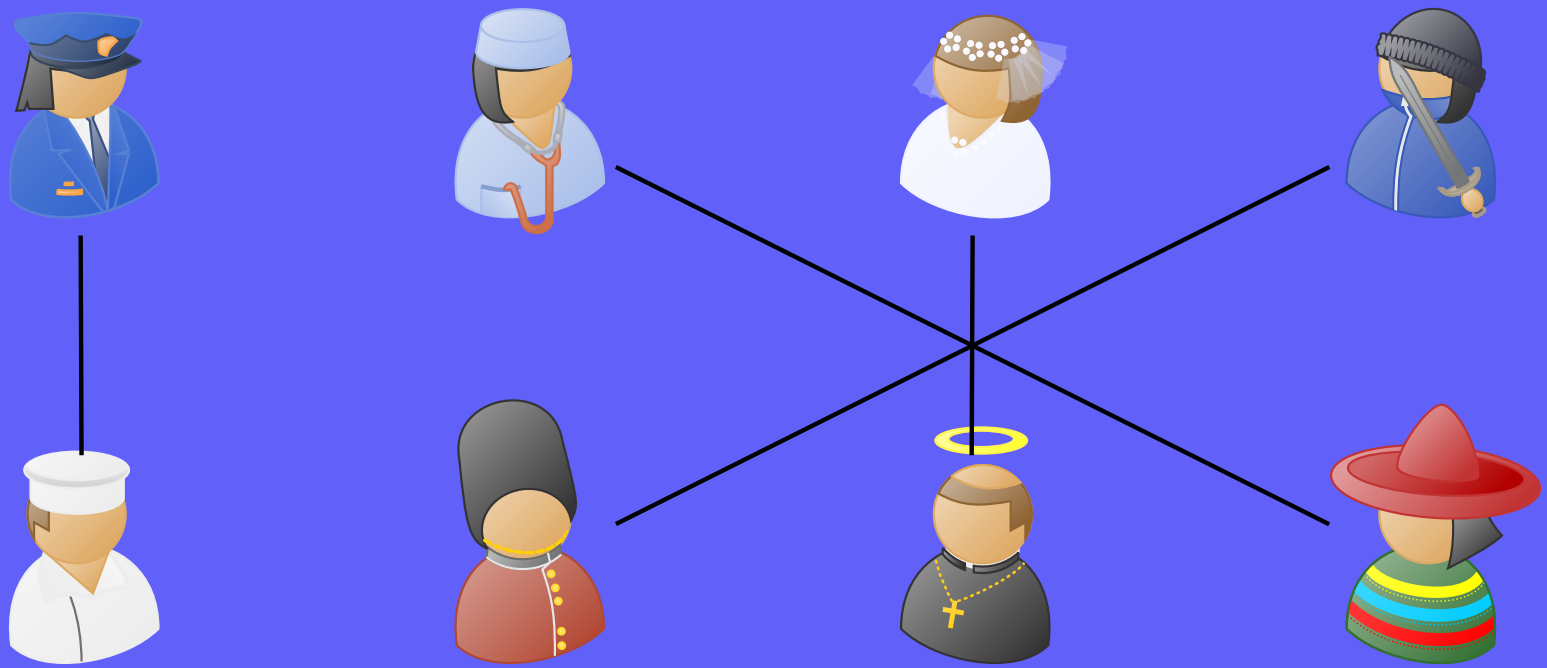
m_1 m_2 m_3 m_4

$w_1 > w_2 > w_3 > w_4$ $w_1 > w_4 > w_3 > w_2$ $w_1 > w_3 > w_4 > w_2$ $w_1 > w_3 > w_2 > w_4$

Stable marriage problem

$m_3 > m_2 > m_1 > m_4$ $m_4 > m_1 > m_2 > m_3$ $m_3 > m_4 > m_1 > m_2$ $m_1 > m_2 > m_3 > m_4$

w_1 w_2 w_3 w_4



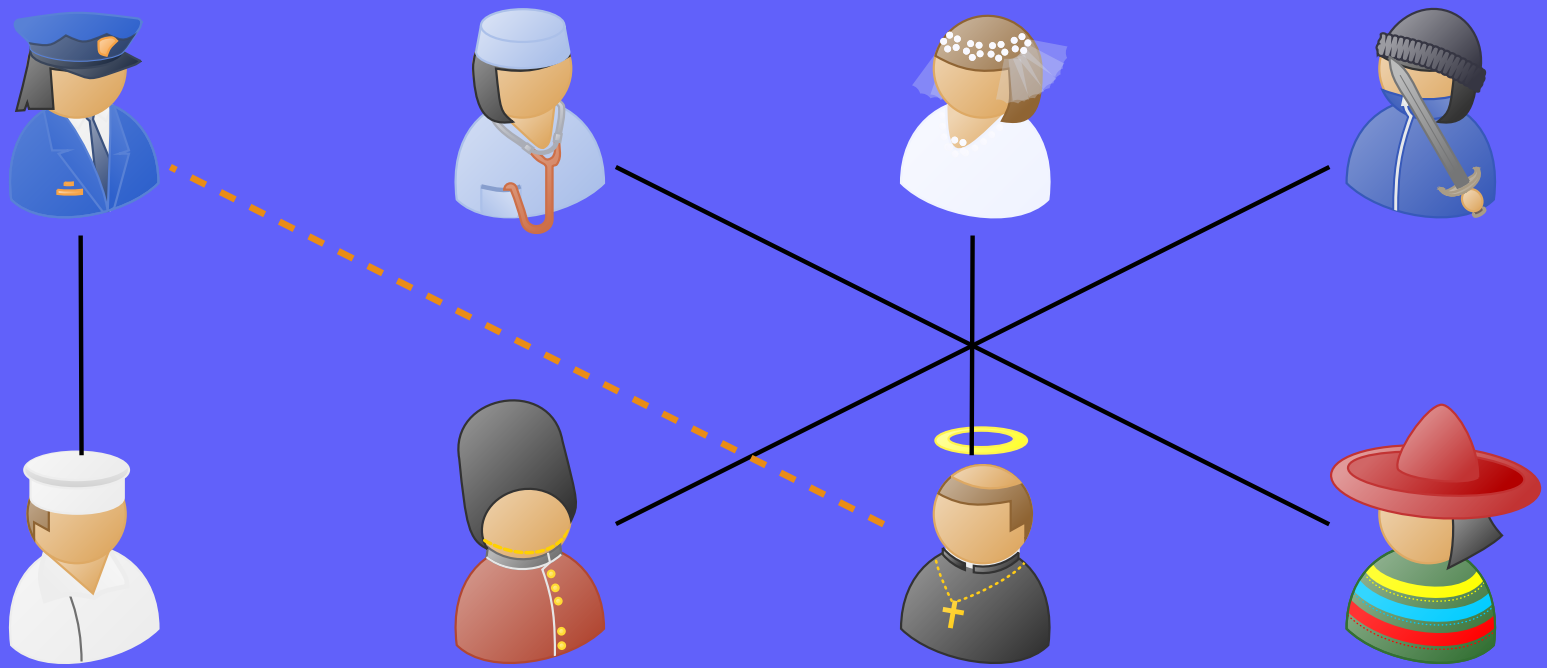
m_1 m_2 m_3 m_4

$w_1 > w_2 > w_3 > w_4$ $w_1 > w_4 > w_3 > w_2$ $w_1 > w_3 > w_4 > w_2$ $w_1 > w_3 > w_2 > w_4$

Stable marriage problem

$m_3 > m_2 > m_1 > m_4$ $m_4 > m_1 > m_2 > m_3$ $m_3 > m_4 > m_1 > m_2$ $m_1 > m_2 > m_3 > m_4$

w_1 w_2 w_3 w_4



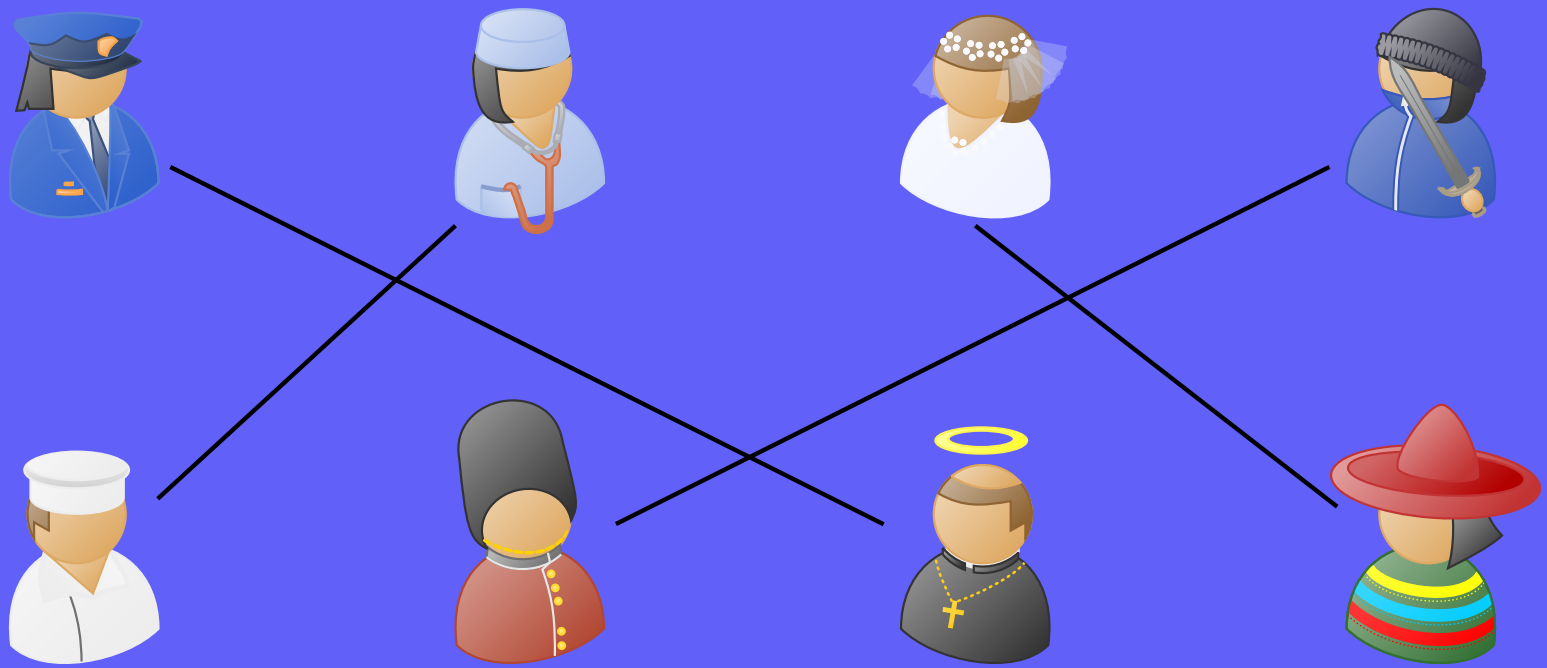
m_1 m_2 m_3 m_4

$w_1 > w_2 > w_3 > w_4$ $w_1 > w_4 > w_3 > w_2$ $w_1 > w_3 > w_4 > w_2$ $w_1 > w_3 > w_2 > w_4$

Stable marriage problem

$m_3 > m_2 > m_1 > m_4$ $m_4 > m_1 > m_2 > m_3$ $m_3 > m_4 > m_1 > m_2$ $m_1 > m_2 > m_3 > m_4$

w_1 w_2 w_3 w_4



m_1 m_2 m_3 m_4

$w_1 > w_2 > w_3 > w_4$ $w_1 > w_4 > w_3 > w_2$ $w_1 > w_3 > w_4 > w_2$ $w_1 > w_3 > w_2 > w_4$

Formalization

Problem (Stable matching problem)

Given a set of n men and a set n women, each person ranks all the members of the other set. A match between the men and women where no (*man*, *woman*)-pair, would prefer to be together rather than with their current partner is said to be *stable*.

Formalization

Problem (Stable matching problem)

Given a set of n men and a set n women, each person ranks all the members of the other set. A match between the men and women where no (*man*, *woman*)-pair, would prefer to be together rather than with their current partner is said to be *stable*.

Visual representation of the problem:

- Each men (women) is represented by a vertex
- No edge connects elements from a same set
- Monogamy is assumed for both men and women
- The number of edges equals the number of vertices in a set
- No blocking pair must exist

Solving the problem

Algorithm. (*Gale-Shapley*)

Input : n men and n women, all initially free
Output : n engaged pairs

```
1  while there is a free man do
2      m ← select a man; w ←favorite woman m hasn't proposed yet;
3      m proposes to w;
4      if w is free then set (m, w) as an engaged pair;
5      else
6          m′←current date of w;
7          if w prefers m′ over m then m remains free;
8          else
9              set (m, w) as an engaged pair;
10             set m′ as free;
11         end if
12     end if
13 end while
14 return the n engaged pairs
```

Correctness

Theorem

Given two sets of n men and n women, all initially free, Gale-Shapley algorithm returns a stable matching.

Proof. First we prove that algorithm 2.15 returns a perfect matching, i.e. there is no free man or woman.
Suppose there exists a free man who has already proposed to all the women. As a woman who is engaged once remains engaged, to the same man or a new one, it means that all the n women are engaged. Noting that a woman cannot be engaged to more than one man leads to a contradiction. ↯

Lets now assume that the matching is not stable. Then there are two pairs (m, w) and (m', w') such that m prefers w' over w while w' prefers m over m'.

Correctness

By definition, the last proposal of m was to w. If he didn't proposed earlier to w', then he does not prefer w' over w. ↯
If he did propose to w', then he was rejected in favor of another man m''. Therefore the final partner of w' is either m' or a man she prefers over m'. Either way w' does not prefer m to m'. ↯

It follows that Gale-Shapley algorithm returns a stable matching.

□

Correctness

By definition, the last proposal of m was to w. If he didn't proposed earlier to w', then he does not prefer w' over w. ↯
If he did propose to w', then he was rejected in favor of another man m''. Therefore the final partner of w' is either m' or a man she prefers over m'. Either way w' does not prefer m to m'. ↯

It follows that Gale-Shapley algorithm returns a stable matching.

□

Corollary

If each man prefers a different woman then they all end up with their first choice, independently of the women preferences.

Complexity

Theorem

Gale-Shapley algorithm has complexity $\mathcal{O}(n^2)$.

Proof. At each iteration of the `while` loop a man proposes to a woman he has never proposed before. Therefore, for n men, there is a maximum of n^2 proposals.

Since each iteration corresponds to exactly one proposal the `while` loop is applied at most n^2 times.

Hence algorithm 2.15 has complexity $\mathcal{O}(n^2)$. □

Complexity of the Union-Find structure

In order to study the complexity of the Union-Find data structure (1.37) we introduce the following simple results.

Lemma

1. A node that is a root and gets attached to another root will never be a root again.
2. When a node stops being a root its rank remains fixed.
3. As Find travels to the root of the tree the rank of the nodes strictly increases.
4. A node with rank k has at least 2^k nodes in its subtree.
5. Over n elements there are at most $n/2^k$ elements of rank k.

Iterated logarithm

Definition

The *iterated logarithm* function, denoted $\log^*$, is defined by

$$\log^* : \quad \mathbb{R} \longrightarrow \mathbb{N}$$

$$x \longmapsto \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log^* \log_2 x & \text{if } x > 1 \end{cases}$$

In a less abstract way the iterated logarithm of n is the number of times the logarithm function has to be applied in order to get a number smaller than 2.

Iterated logarithm

Definition

The *iterated logarithm* function, denoted $\log^*$, is defined by

$$\log^* : \quad \mathbb{R} \longrightarrow \mathbb{N}$$

$$x \longmapsto \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log^* \log_2 x & \text{if } x > 1 \end{cases}$$

In a less abstract way the iterated logarithm of n is the number of times the logarithm function has to be applied in order to get a number smaller than 2.

Example.

$$\log^* 4 = 1 + \log^* \log_2 4 = 2$$

$$\log^* 16 = 1 + \log^* 4 = 3$$

$$\log^* 536 = \log^* 2^{2^{2^2}} = 4$$

Complexity for m Find

Theorem

The cost of one `Find` in the Union-Find data structure is $\mathcal{O}(\log n)$, while m `Find` operations cost $\mathcal{O}\left((m+n)\log^* n\right)$.

Proof. The rank of the root being bounded by the depth of the tree in the Union-Find data structure it is at most $\log n$.

The problem is now to evaluate the effect of m such operations. Since the path is compressed at each `Find` the complexity of any subsequent `Find` reusing a visited path is decreased.

In fact the running time of a `Find` operation is proportional to (i) the length of the path from a node to the root of the tree it belongs to and (ii) m.

Complexity for m Find

We now divide the nodes into "blocks", such that nodes of similar ranks are on a same block. To formerly define a block we introduce the following recursive sequence

$$\begin{cases} T_0 = 1 \\ T_i = 2^{T_{i-1}}, \quad i > 0. \end{cases}$$

The blocks are then defined as the sets

$$\begin{cases} B_0 = \{0\}, \ B_1 = \{1\}, \\ B_i = [T_{i-1}, T_i - 1], \qquad i > 1. \end{cases}$$

Observing the blocks we notice that the maximum number of blocks is $\log^* n$.

Complexity for m Find

Moreover as there is a maximum of $n/2^k$ elements of rank k (lemma 2.19), the number of nodes in the i-th block is at most

$$\begin{aligned}\sum_{j=T_{i-1}}^{T_i-1} \frac{n}{2^j} &= n \sum_{j=T_{i-1}}^{2^{T_{i-1}}-1} \frac{1}{2^j} \\ &\leq \frac{2n}{2^{T_{i-1}}} \\ &= \frac{2n}{T_i}.\end{aligned} \tag{2.1}$$

The idea is to evaluate the cost of the `Find` while traveling from a node to its root. In such a case we traverse nodes which are (i) in the same block (ii) in a different block or (iii) directly connected to the root.

Complexity for m Find

The simplest case is (iii) as only one step is required, implying an $\mathcal{O}(m)$ complexity for the m Find.

Note that (ii) was evaluated earlier when we observed that the maximum number of blocks is $\log^* n$. Therefore the complexity of (ii) is $\mathcal{O}(m \log^* n)$.

Finally for case (i) the path starting at a node x goes through at most $T_i - 1 - T_{i-1}$ ranks, and from (2.1) there are less than $2n/T_i$ such elements x in a block. Thus the total number of times the rank changes in a block is

$$\begin{aligned}(T_i - 1 - T_{i-1})\frac{2n}{T_i} &\leq T_i \frac{2n}{T_i} \\ &= 2n.\end{aligned}$$

Complexity for m Find

Remembering that the maximum number of blocks is $\log^* n$, the `Find` operations generate an overall of at most $\mathcal{O}(n \log^* n)$ internal links.

Hence the time spent on m operations is given by the total time spent on cases (i) to (iii), that is $\mathcal{O}\left((m+n)\log^* n\right)$. □

For long this has been the tightest proven upper bound. However a better result was proven in the 1970's. It bounds the complexity by using the inverse Ackerman's function, which is growing even slower than the $\log^*$ function.

Ackerman's function

Definition (Ackerman's function)

For any two positive integers m and n, Ackerman's function is recursively defined by

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Ackerman's function

Definition (Ackerman's function)

For any two positive integers m and n, Ackerman's function is recursively defined by

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Example.

$$A(0,0) = 1$$
$$A(1,1) = A(0, A(1,0)) = A(0,2) = 3$$
$$A(2,2) = A(1, A(2,1)) = A(1,5) = 7$$
$$A(3,3) = A(2, A(3,2)) = A(2,29) = 61$$
$$A(4,4) = A(3, A(4,3)) = 2^{2^{2^{65536}}} - 3$$

Inverse Ackerman's function

For $m = n$ one defines the inverse of Ackerman's function $\alpha(x)$. As Ackerman's function is extremely fast-growing its inverse is an extremely-slow growing function which never increases beyond 4 or 5 for any practical value.

Theorem

The amortized time for a sequence of m `GenSet`, `Union`, and `Find` operations, n of which are `GenSet`, can be performed in time $\mathcal{O}(m\alpha(n))$, where $\alpha(n)$ is the inverse Ackerman's function.

Inverse Ackerman's function

For $m = n$ one defines the inverse of Ackerman's function $\alpha(x)$. As Ackerman's function is extremely fast-growing its inverse is an extremely-slow growing function which never increases beyond 4 or 5 for any practical value.

Theorem

The amortized time for a sequence of m `GenSet`, `Union`, and `Find` operations, n of which are `GenSet`, can be performed in time $\mathcal{O}(m\alpha(n))$, where $\alpha(n)$ is the inverse Ackerman's function.

Remark. It was recently proven that this upper bound is asymptotically optimal, that is the complexity of the Union-Find structure is $\Omega(\alpha(n))$.

Complexity of Sort and Count

The Sort and Count algorithm (1.44) solves the Counting inversions problem (1.40).

For the merge part:

- Each iteration of the `While` takes constant time
- Elements that are added are never added again
- There is a maximum of n element

The time spent for the merge part is $\mathcal{O}(n)$.

Complexity of Sort and Count

The Sort and Count algorithm (1.44) solves the Counting inversions problem (1.40).

For the merge part:

- Each iteration of the `While` takes constant time
- Elements that are added are never added again
- There is a maximum of n element

The time spent for the merge part is $\mathcal{O}(n)$.

For the sort part the divide and conquer strategy is applied:

- Divide the input into two pieces of equal size
- Recursively solve the two separate problems
- Combine the two results

The time for division and recombination is linear in the size of the input.

Divide and conquer

Let $S(n)$ be the worst-case run time for sorting an instance of size n.

Simple complexity analysis:

- Time spent to divide into two pieces: $\mathcal{O}(n)$
- Time spent to solve each piece: $S(n/2)$
- Combine the results: $\mathcal{O}(n)$

Divide and conquer

Let $S(n)$ be the worst-case run time for sorting an instance of size n.

Simple complexity analysis:

- Time spent to divide into two pieces: $\mathcal{O}(n)$
- Time spent to solve each piece: $S(n/2)$
- Combine the results: $\mathcal{O}(n)$

Hence, the running time satisfies the recurrence relation

$$\begin{cases} S(2) \leq c & \text{for some constant } c \\ S(n) \leq 2S(n/2) + cn & \text{if } n > 2 \end{cases}$$

The Master Theorem

Let A be an algorithm whose running time is described by a recurrence of the form $T(n) = aT(n/b) + f(n)$, with $a \geq 1$, $b > 1$ two constants, and $f(n)$ an asymptotically positive function. This relation corresponds to A dividing the initial problem of size n into a sub-problems of size n/b each. While it takes $T(n/b)$ to recursively solve each of the a sub-problems $f(n)$ corresponds to the time spent splitting them and combining their results.

Remark. For the sake of simplicity we assume n/b to be an integer, and more generally n to be a power of b. Although it is often not true, this simplifies the discussion while having no impact on the asymptotic behavior of the recurrence.

The Master Theorem

Theorem (Master theorem)

Let $a \geq 1$, $b > 1$, be two constants, $f(n)$ be a function, and $T(n) = aT(n/b) + f(n)$ be a recurrence relation over the positive integers. Then the asymptotic bound on $T(n)$ is given by

$$T(n) = \begin{cases} \Theta(n^{\log_b a} \log n) & \text{if } f(n) = \Theta(n^{\log_b a}); \\ \Theta(n^{\log_b a}) & \text{if } f(n) = \mathcal{O}(n^{\log_b a - \varepsilon}),\ \varepsilon > 0; \\ \Theta(f(n)) & \text{if } f(n) = \Omega(n^{\log_b a + \varepsilon}),\ \varepsilon > 0,\ n \text{ large enough, and } af(n/b) \leq cf(n),\ c < 1; \end{cases}$$

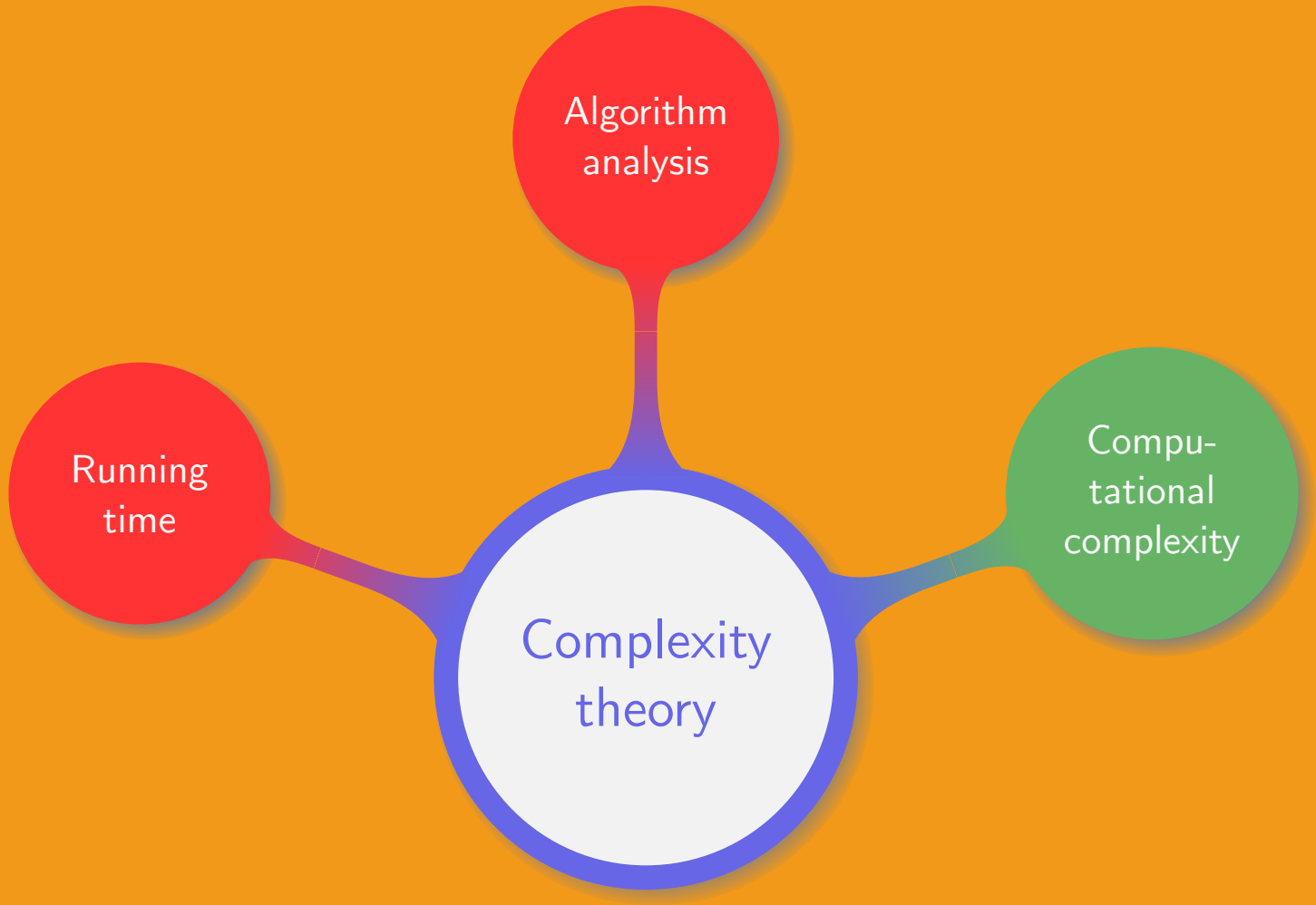
Back to Sort and Count

Summary of the complexity:

- Applying the Master theorem (2.31) to the sort part yields a complexity in $\mathcal{O}(n \log n)$
- The complexity of the merge part is in $\mathcal{O}(n)$

The overall time complexity of Sort and Count is $\mathcal{O}(n \log n)$

Chapter organisation



Computational problem

So far we have presented a few problems as well as some algorithms to solve them. We have also introduced the basics to study the complexity of those algorithms.

We now formalize these ideas through the following definitions.

Definitions

1. A *computational problem* is a question or set of questions that a computer might be able to solve.
2. The study of the solutions to computational problems composes the field of *Algorithms*.
3. *Computational complexity* attempts to classify algorithms depending on their speed or memory usage.

Taxonomy of computational problems

A *decision problem* is a computational problem admitting exactly one of the two answers, "Yes" or "No", to the question.

Example. Let n be an integer, is n prime?

A *search problem* is a computational problem where the answer is an arbitrary string.

Example. Let n be an integer, find all the primes less than n.

A *counting problem* is a computational problem where the answer is the number of solutions to a corresponding search problem.

Example. Let n be an integer, count the number of primes less than n.

Taxonomy of computational problems

Am *optimization problem* is a computational problem where the answer is the best solution, with respect to some parameters, to a corresponding search problem.

Example. For n a non-prime integer, find the largest prime factor of n.

A *function problem* is a computational problem admitting exactly one answer for every input. The answer is more complex than in the case of a decision problem.

Example. Given a list of cities and the distance between each pair, find the shortest route passing through all the cities exactly once and returning to the first visited city.

Relation between computational problems

An obvious observation is that counting and optimization problems are closely related to search problems.

As a less obvious observation one can notice that any optimization problem can be transformed into a decision problem.

Relation between computational problems

An obvious observation is that counting and optimization problems are closely related to search problems.

As a less obvious observation one can notice that any optimization problem can be transformed into a decision problem.

Example. Optimization problem: let G be a graph and v_1, v_2 be two vertices in G. Find the shortest path between v_1 and v_2.

A corresponding decision problem: let G be a graph and v_1, v_2 be two vertices in G. Is there a path from v_1 to v_2 that goes through less than 5 edges?

Relation between computational problems

Similarly any function problem can be transformed into a decision problem. If the decision problem can be solved then so is its corresponding function problem. However the computational cost is not always preserved; for instance a function problem might be solved by an exponential time algorithm while its corresponding decision problem can be solved in polynomial time.

Conversely decision problems can be converted into function problems by computing the characteristic function of the set associated with the decision problem.

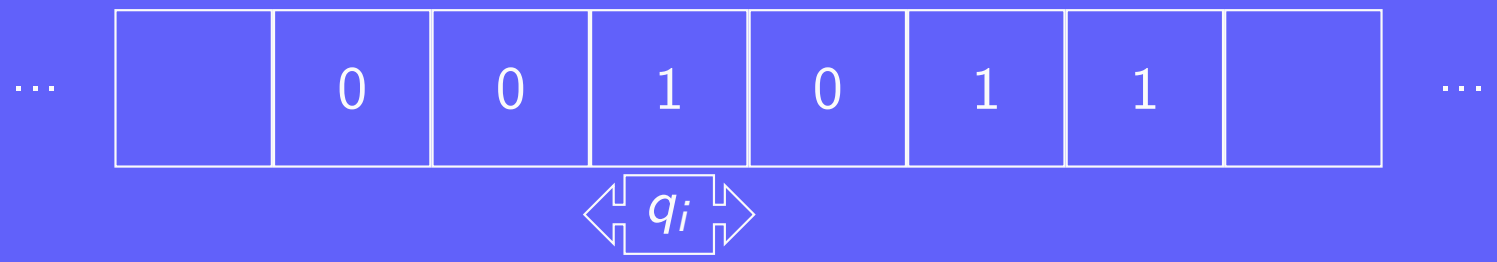
Decision problems play a central role in computability and complexity theories. In order to study them in more details we first need to setup a more precise computational model.

Turing machine

...an unlimited memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol, and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings.

Turing – *Intelligent Machinery*

Turing machine



A Turing machine can be use to model modern computers:

- It is composed of an infinite tape
- The tape is divided into cells
- Each cell contains a symbol taken from an alphabet or a blank
- The tape is read or written, cell by cell, by a head
- At any time the device is in a state q_i with $i \in \mathbb{N}$

Elementary operation

Let Σ be the alphabet of symbols and Q be the set of the states. Any elementary operation is determined by the symbol read and the state $q_i \in Q$.

Three basic cases can occur:

- The symbol read is replaced by another symbol from $\Sigma \cup \{b\}$, where b represents a blank
- The head moves to the left, right, or stands still
- The machine transitions from state q_i to state q_j

This can be summarized by the following map:

$$M : (\Sigma \cup \{b\}) \times Q \longrightarrow (\Sigma \cup \{b\}) \times \{-1, 0, 1\} \times Q$$

Deterministic computation

A deterministic computation is defined by (i) an initial state q_0 and (ii) a finite sequence of elementary operations.

At the end of the sequence the tape contains an integer x written over a number of cells, all the other cells being set to b (blank). The state of the tape, x, provides the result.

Deterministic computation

A deterministic computation is defined by (i) an initial state q_0 and (ii) a finite sequence of elementary operations.

At the end of the sequence the tape contains an integer x written over a number of cells, all the other cells being set to b (blank). The state of the tape, x, provides the result.

Definition

A function $f : \Sigma^* \longrightarrow \Sigma^*$ is said to be *Turing computable* if there exists a Turing machine M which returns $f(x)$ for any input x.

Remark. Church proved that any computation, in a physical sense, can be performed on a Turing machine. This is however not the case anymore for quantum computers which cannot be modeled by a Turing machine. Its quantum counterpart is called *Quantum Turing machine*.

The RAM model

The RAM consists of:

- A control unit: containing a program and a program register pointing to the instruction to be executed
- An arithmetic unit: executes all the arithmetic operations
- A memory: divided into cells, each containing an integer
- An input unit: an input tape divided into cells and a head which reads the input
- An output unit: an output tape divided into cells and a head which writes the output

A program in the RAM model

The initial configuration of a RAM:

- All the cells are set to 0, besides the first n input cells
- The program register contains 1
- The first n cells contain the value from the n input cells

A computation consists in performing a sequence of configuration based on the program.

A program is composed of operands stored in the memory and onto which instructions can be run (e.g. `LOAD`, `STORE`, `+`, `-`, $\times$, `/`, `READ`, `WRITE`, `JUMP`, `JZERO`, `JGE`, `HALT`, `ACCEPT`, `REJECT`).

RAM vs. Turing machine

The RAM represents a better model for modern computer than a Turing machine. However those two models are equivalent.

Theorem

- For every Turing machine M, there exists a program P for RAM that simulates M.
- For a program P of RAM, there exists a Turing machine with five tapes such that P and M behave the same.

As we have briefly introduced the most common computational models we can now formalize the idea of complexity, and then investigate decision problems.

Formalizing complexity

Definitions

Let Σ be an alphabet and M be a Turing machine.

1. Given an input $x \in \Sigma^*$ for M, the number of operations $t_M(x)$ necessary to perform the computation is called the *length of the computation*.
2. The function

$$T_M : \mathbb{N} \longrightarrow \mathbb{N}$$

$$x \longmapsto \max_{|x|} t_M(x)$$

defines the *time complexity* for M.

Formalizing complexity

Definitions

Let Σ be an alphabet and M be a Turing machine.

1. Given an input $x \in \Sigma^*$ for M, the number of operations $t_M(x)$ necessary to perform the computation is called the *length of the computation*.
2. The function

$$T_M : \mathbb{N} \longrightarrow \mathbb{N}$$

$$x \longmapsto \max_{|x|} t_M(x)$$

defines the *time complexity* for M.

Remark. Since the maximum of the length of the computation is considered this definition corresponds to the worst case complexity (2.4) with $|x|$, the size of the input x equal to n.

Decision problems

Definitions

1. Let f be a Turing computable function, and M be that Turing machine. If there exists a polynomial P such that for any input x, $T_M(x) \leq P(x)$, then M is called a *deterministic polynomial algorithm*.
2. Let $f : \Sigma^* \to \{0, 1\}$ be a Turing computable function and L be the preimage of $\{1\}$ under f, i.e. $f^{-1}(1)$. Then L is called a *language* and f defines a *decision problem* P. One says that the Turing machine *computes* f, *solves* P, or *decides* L.

The $\mathcal{P}$ and $\mathcal{NP}$ classes

Definitions

1. The set of the decision problems which can be solved by a deterministic polynomial algorithm defines the *class* $\mathcal{P}$.
2. A decision problem Π is computable by a non deterministic polynomial algorithm if and only if there exists a Turing machine M and a polynomial P such that
   i. $x \in L(\Pi)$ if and only if there exists $y \in \Sigma^*$, such that M computes 1 when it has input x in the cells 1 to $|x|$ and y in the cells -1 to $-|y|$.
   ii. For all x in $L(\Pi)$ there exists y such that M computes 1 in time less than $P(|x|)$.

   The set of all such decision problems defines the class $\mathcal{NP}$.

Explanations

Informal view:

- Class $\mathcal{P}$: set of decision problems solvable in polynomial time
- Class $\mathcal{NP}$: set of decision problems which (i) can be solved in polynomial time, assuming a *certificate* $y \in \Sigma^*$, is known and (ii) have `True` as answer.
  The certificate is often taken to be the solution and it means that the solution can be verified in polynomial time.
- Class co-$\mathcal{NP}$: set of decision problems which (i) can be solved in polynomial time, assuming a *certificate* $y \in \Sigma^*$, is known and (ii) have `False` as answer.
  The certificate is often taken to be the solution and it means that the solution can be verified in polynomial time.
- Any problem in $\mathcal{P}$ is also in $\mathcal{NP}$ and co-$\mathcal{NP}$.

Complexity reduction

Definitions

1. Let P_1 and P_2 be two decision problems. One says that P_1 can be *reduced in polynomial time* to P_2 and writes $P_1 \propto P_2$, if there exists a function f, computable in polynomial time, such that $x \in L(P_1)$ if and only if $f(x) \in L(P_2)$.
2. A problem Π is $\mathcal{NP}$*-hard* if and only if for all P in $\mathcal{NP}$, P can be reduced in polynomial time to Π.
3. A problem Π is $\mathcal{NP}$*-complete* if and only if Π is (i) in $\mathcal{NP}$ and (ii) $\mathcal{NP}$-hard.

Explanations

Informal view:

- Saying that P_1 can be reduced in polynomial time to P_2 means that P_1 is not any harder than P_2. In fact any instance of P_1 can be transformed, in polynomial time, into an instance of P_2.
- An $\mathcal{NP}$-hard problem Π is at least as hard as the hardest problem in $\mathcal{NP}$. Note that Π does not need to belong to $\mathcal{NP}$, it could for instance be a search problem or an optimization problem.
- An $\mathcal{NP}$-complete problem Π must belong to $\mathcal{NP}$ and is as hard as the hardest problem in $\mathcal{NP}$. In other words if we can solve Π then we can solve any other problem in $\mathcal{NP}$, deriving the solution from the one of Π, in polynomial time.

Halting problem

Problem (Halting problem)

Given the description of a Turing machine M as well as its initial input, decide whether M will halt or run forever.

The problem consists in constructing a Turing machine which would be able to decide whether or not another Turing machine would complete its task. From a simple perspective, if the machine "quickly" completes its task then we know it. Otherwise we have no idea whether it will halt later or never stop.

Turing proved that the halting problem is *undecidable*, i.e. it is impossible to construct an algorithm which always returns the right "yes" or "no" answer. Therefore it does not belong to $\mathcal{NP}$.

Boolean Satisfiability Problem

Problem (Boolean Satisfiability Problem (SAT))

Given a boolean function $(x_1, \cdots, x_n) \mapsto f(x_1, \cdots, x_n)$, where the x_i, $0 \leq i \leq n$ are in $\{0, 1\}$, and f is an expression constructed from the x_i and the boolean symbols $\neg$, $\vee$, and $\wedge$, is there a value for the n-tuple $\langle x_1, \cdots, x_n \rangle$ such that $f(x_1, \cdots, x_n)$ is `True`?

Example. The expression $x_1 \wedge \neg x_2$ is satisfiable since taking $x_1 = \texttt{True}$ and $x_2 = \texttt{False}$ yields $\texttt{True} \wedge \texttt{True} = \texttt{True}$.

On the other hand the expression $x_1 \wedge \neg x_1$ is not satisfiable since it will always evaluate as `False`, independently of the choice of x_1.

Boolean Satisfiability Problem

Theorem (Cook)

SAT is $\mathcal{NP}$-complete.

Sketch of proof. First, SAT is in $\mathcal{NP}$ as any instance can be verified in polynomial time on a Turing machine.

Second, any $\mathcal{NP}$ problem P can be verified in polynomial time on a Turing machine M. Therefore for each input to M, it is possible to construct a boolean formula checking (i) every step of the computation, (ii) if M halts, and (iii) M returns "Yes". Such a boolean expression will be satisfied if and only if those three conditions are met, that is if P is solved.

Finally as it can be proven that the boolean expression can be constructed in polynomial time, SAT is $\mathcal{NP}$-complete. □

SAT and the Halting problem

Theorem

The Halting problem is $\mathcal{NP}$-hard.

Sketch of proof. The basic idea consists in reducing SAT (2.53) to the Halting problem (2.52).

It suffices to transform SAT into a Turing machine that tries all possible assignments of truth values. If a solution is found, then halt and otherwise start an infinite loop.

Since SAT is $\mathcal{NP}$-complete it means that any problem P in $\mathcal{NP}$ can be reduced in polynomial time to the Halting problem. $\square$

True quantified Boolean formula

Problem (True Quantified Boolean Formula (TQBF))

A quantified boolean formula is a formula that can be written in the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \phi(x_1, x_2, \cdots, x_n),$$

where each of the Q_i is one of the quantifier $\exists$ or $\forall$.

Calling L the language composed of the `True` quantified boolean formulae, decide L.

Example. The quantified boolean formula

$$\forall x_1 \exists x_2 \forall x_3 \left((x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge x_3)\right)$$

is `False`.

True quantified Boolean formula

Problem (True Quantified Boolean Formula (TQBF))

A quantified boolean formula is a formula that can be written in the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \phi(x_1, x_2, \cdots, x_n),$$

where each of the Q_i is one of the quantifier $\exists$ or $\forall$.

Calling L the language composed of the `True` quantified boolean formulae, decide L.

Example. The quantified boolean formula

$$\forall x_1 \exists x_2 \forall x_3 \left((x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge x_3)\right)$$

is `False`.

Observe that x_3 must be `True` in order for the formula to evaluate as `True`. However x_3 is a universally quantifiable variable.

True quantified Boolean formula

Theorem

TQBF can be solved in exponential time and polynomial space.

Sketech of proof. The idea consists in exhibiting a polynomial-space algorithm that decides whether any given Quantified Boolean Formula (QBF) is true.

Consider a general input

$$\Psi = Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \phi(x_1, x_2, \cdots, x_n),$$

where ϕ has m clauses. We want to use the same argument as in example 2.56, i.e. if Q_i is $\forall$ then both formulae, with x_i replaced by 0 and 1, must evaluate as `True`; if Q_i is $\exists$ then only one of them is required to be `True`.

True quantified Boolean formula

This is achieved by "pilling off" the quantifiers one by one. Starting with Q_1, one evaluates both

$$\Psi_{1_0} = Q_2 x_2 \cdots Q_n x_n \phi(0, x_2, \cdots, x_n) \text{ and}$$
$$\Psi_{1_1} = Q_2 x_2 \cdots Q_n x_n \phi(1, x_2, \cdots, x_n).$$

Then compute the logical value of $\psi_{1_0} \vee \psi_{1_1}$ or $\psi_{1_0} \wedge \psi_{1_1}$, depending whether Q_1 is $\exists$ or $\forall$.

Following this strategy all the quantifiers are recursively removed and it then suffices to solve a SAT problem.

The computational cost of such a strategy is very high since at each recursive call the problem is transformed into two new linearly smaller sub-problems, resulting in a complexity of $\mathcal{O}(2^n)$.

True quantified Boolean formula

We now consider the space necessary to solve TQBF.

As Ψ_i is computed either from $\Psi_{(i+1)_0} \wedge \Psi_{(i+1)_1}$ or $\Psi_{(i+1)_0} \vee \Psi_{(i+1)_1}$, the two instances of $\Psi_{(i+1)}$ can be computed sequentially. Therefore they can use the same memory space.

However at each level of recursion the indices must be saved. This incurs an $\mathcal{O}(\log n)$ overhead, and as a result if computing Ψ_{i+1} requires space S_{i+1} then evaluating Ψ_i requires

$$S_i = S_{i+1} + \mathcal{O}(\log n).$$

Hence Ψ can be reached in $\mathcal{O}(n \log n)$ space, that is in polynomial space.

□

PSPACE completeness

Definition

1. Let SPACE denote the set of all the decision problems which can be solved by a Turing machine in $\mathcal{O}(s(n))$ space for some function s of the input size n. Then PSPACE is defined as

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k).$$

2. A problem Π is PSPACE-*complete* if and only if (i) Π is in PSPACE and (ii) for all P in PSPACE, P can be reduced in polynomial space to Π.

PSPACE completeness

Definition

1. Let SPACE denote the set of all the decision problems which can be solved by a Turing machine in $\mathcal{O}(s(n))$ space for some function s of the input size n. Then PSPACE is defined as

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k).$$

2. A problem Π is PSPACE-*complete* if and only if (i) Π is in PSPACE and (ii) for all P in PSPACE, P can be reduced in polynomial space to Π.

Theorem

TQBF is PSPACE-complete and in particular it is $\mathcal{NP}$-hard.

Hilbert's tenth problem

Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

Hilbert – *List of Hilbert's problem list of 1900*

Hilbert's tenth problem

Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

Hilbert – *List of Hilbert's problem list of 1900*

Problem (Hilbert's tenth problem)

Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients, decide whether the equation is solvable in rational integers.

Beyond Hilbert's tenth problem

After a 21 years long quest Matiyasevich proved that Hilbert's tenth problem was undecidable, that is there is no algorithm which can determine whether any given Diophantine equation is solvable in rational integers.

A side result however states that one can exhibit a Diophantine equation with no solution, but such that it is impossible to prove it.

The idea is to number all the Diophantine equations, $E_1, \cdots, E_n, \cdots$ following some set of axioms such as Peano arithmetic or Zermelo-Fraenkel set theory. The number of axioms and logical operators being finite, one can construct all the possible proof $P_1, \cdots, P_n \cdots$.

At stage 1, consider E_1 and check if:

- There exists a solution for the positive integers less than 1
- P_1 is a proof that E_1 has no solution

Beyond Hilbert's tenth problem

At stage k, consider $E_1, \cdots, E_k$ and check if:

- There exists a solution for the positive integers less than k
- $P_1, \cdots, P_k$ proves one of $E_1, \cdots, E_k$

Then there is at least one Diophantine equation which has neither a solution nor a proof.

This conclusion is drawn from *Gödel incompleteness Theorem* which states that for any axiomatic system built over the Peano arithmetic there is an undecidable problem.

In order to relate *Gödel incompleteness Theorem* to Turing machine we informally define what it means for a set of axioms to be *complete* and *consistent*.

Gödel and Turing

A set of axioms where any statement, built from it, or its negation can be proven in this set of axioms is said to be *complete*. It is *consistent* if it is impossible to prove both a statement and its negation in the set of axioms.

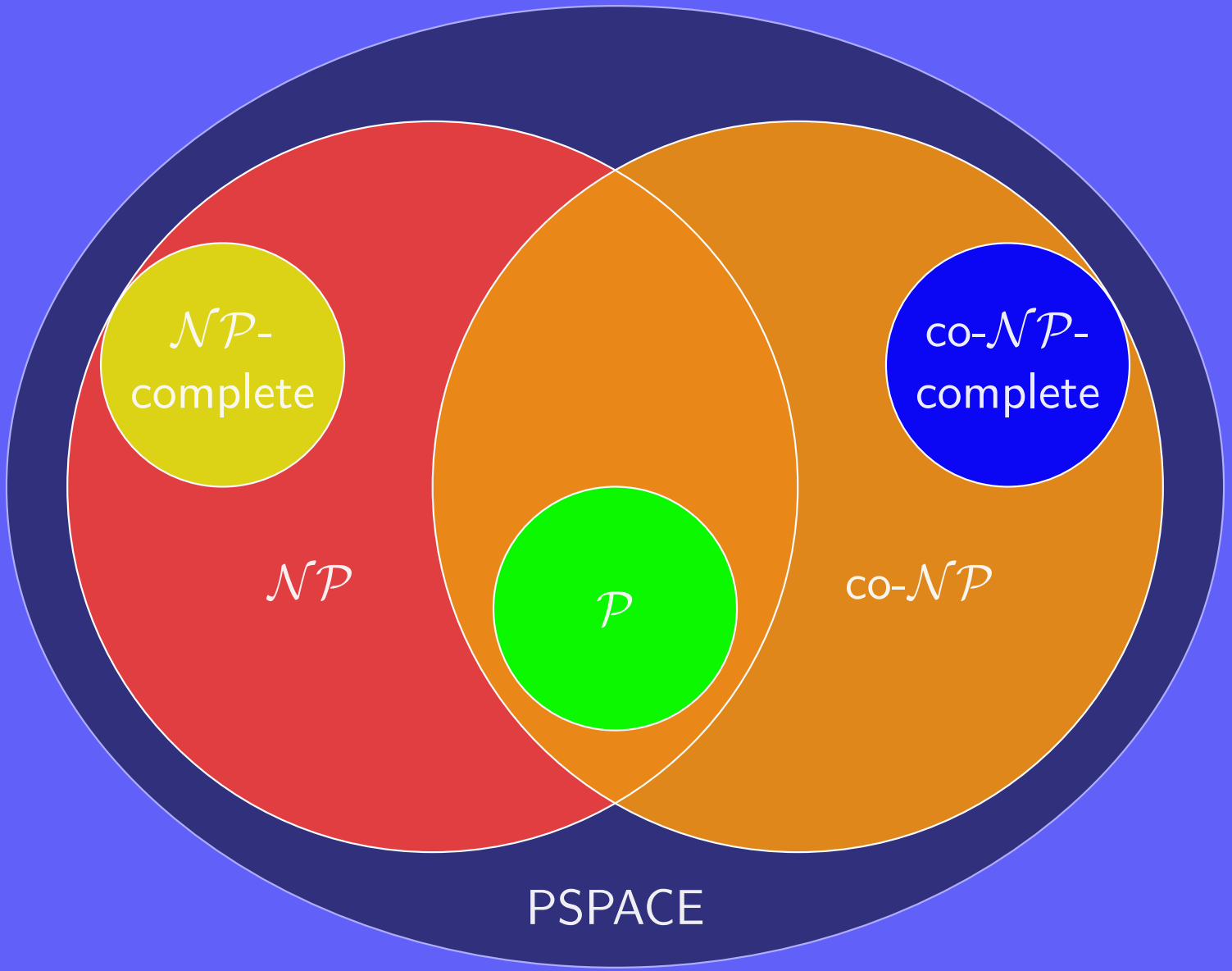
Assume we have a complete and consistent system F, powerful enough to reason about Turing machines. Taking a Turing machine M we can determine whether M halts by enumerating all the possible proofs in F, until one proof states that M halts or runs forever.

The completeness ensures of the correctness of the result while the consistency confirms the truthfulness of the conclusion.

We have proven that given F we can decide the Halting problem (2.52), which is known to be undecidable.

Therefore such a complete and consistent system F cannot exist.

Complexity diagram



A few last remarks

Results known to be true:

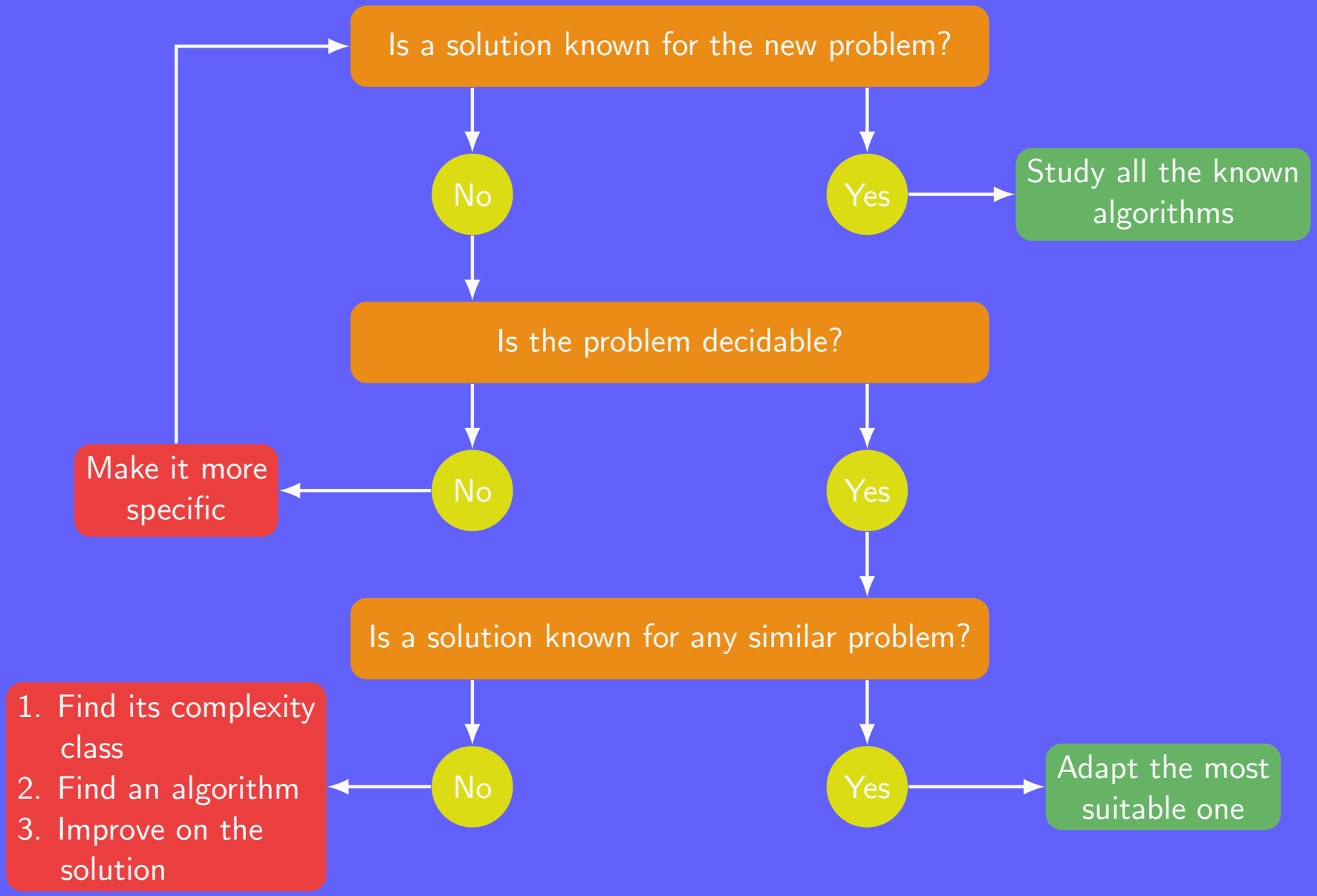
- $\mathcal{P} \subset \mathcal{NP}$
- $\mathcal{NP} \subset$ PSPACE
- PSPACE = NPSPACE (Savitch's theorem)

Results believed to be true:

- $\mathcal{P} \neq \mathcal{NP}$
- $\mathcal{NP} \neq$ PSPACE

More results on computational complexity theory are available at the Complexity Zoo.

Connection to algorithmic



Conjunctive Normal Form

We now provide a formal reduction proof as an example of how to proceed to evaluate the complexity class of a given problem. We start with the following definition.

Definition

A boolean formula is said to be in *Conjunctive Normal Form (CNF)* if it is written as the conjunction of disjunctive clauses.

As a reminder the truth table of the conjunction and disjunction of A and B is

A	B	$A \wedge B$	$A \vee B$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

3-SAT

Problem (Satisfiability with 3 literals per clause (3-SAT))

Given a Boolean formula in CNF where each clause contains exactly three literals is there a truth assignment which satisfies all the clauses.

Theorem

3-SAT is $\mathcal{NP}$-complete.

Proof. Clearly 3-SAT is in $\mathcal{NP}$, for it is a particular case of SAT.

To prove that 3-SAT is $\mathcal{NP}$-hard we will show that being able to solve it implies being able to solve SAT, which by Cook theorem (2.54) is known to be $\mathcal{NP}$-hard.

3-SAT

Assuming CNF, we want to transform any instance of SAT into an instance of 3-SAT. Therefore we need to consider the cases where SAT has clauses with (i) one, (ii) two or (iii) more than three literals. Note that the case where there are exactly three clauses is already 3-SAT so no work is necessary.

(ii) The most simple case is when there are two literals, organised in a unique disjunctive clause $x_1 \vee x_2$ denoted (x_1, x_2). It then suffices to consider the pair of (x_1, x_2, u) and $(x_1, x_2, \neg u)$, where u is a newly added literal.

(i) Similarly in the case of a single literal x we convert it into a pair of literals (x, u_1) and $(x, \neg u_1)$, for a new literal u_1. It then suffices to apply the same strategy as above and create the four literals

$$(x, u_1, u_2),\ (x, u_1, \neg u_2),\ (x, \neg u_1, u_2),\ (x, \neg u_1, \neg u_2).$$

3-SAT

The general idea behind (i) and (ii) is to add new literals which do not impact the satisfiability of the problem.

(iii) The case of a clause with more than three literals $(x_1, \cdots, x_n)$ can be treated by adding some new literals such as to create a chain of clauses where each of them has exactly three literals.

$$(x_1, x_2, u_3), (\neg u_3, x_3, u_4), (\neg u_4, x_4, u_5), \cdots, (\neg u_{n-1}, x_{n-1}, x_n).$$

The question is now to know if this change alters the satisfiability of the original clause.

Assume the original clause evaluates as `True`. Then at least one literal x_i must be `True`. Setting u_j to `True` for all $j \leq i$ and `False` for $j > i$ preserves the satisfiability of the clause.

3-SAT

If now the original clause evaluates as `False` we need to ensure that the altered version cannot be satisfied. If it is originally `False` then all the x_i must be `False`.

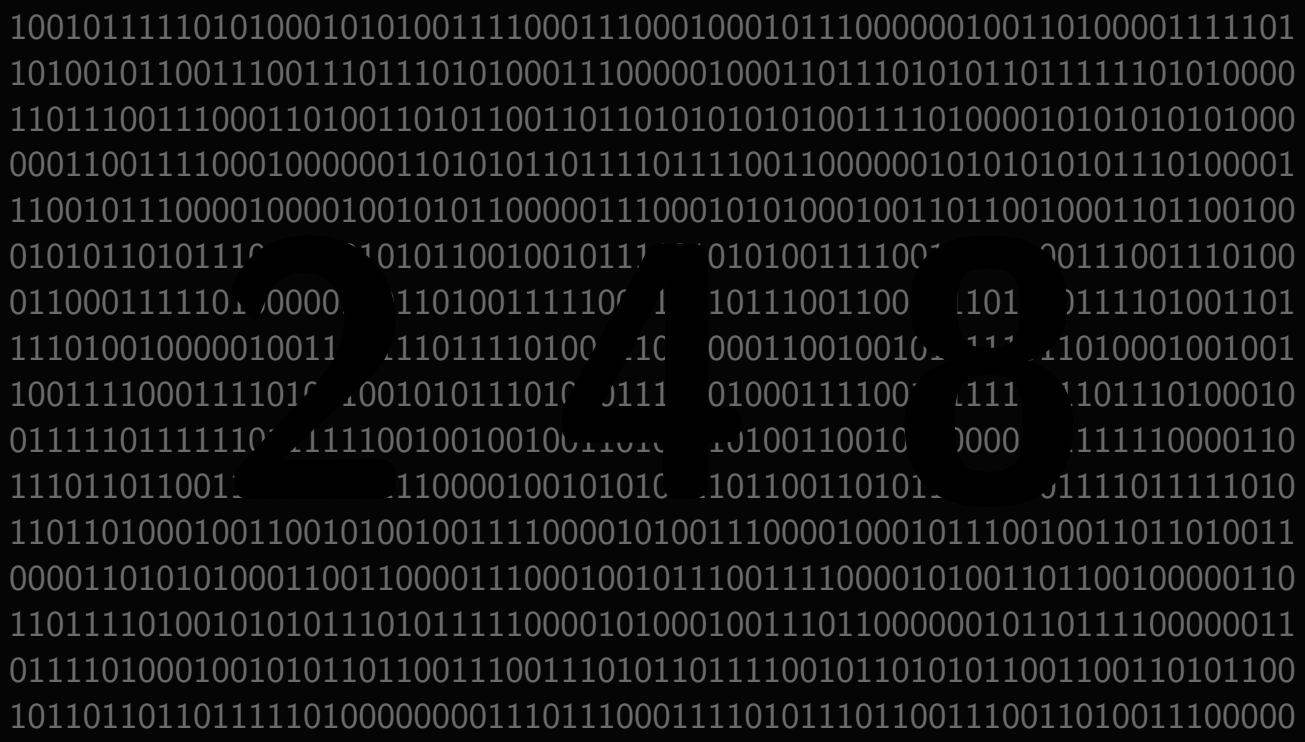
Therefore in the altered version the last clause can only be `True` if u_{n-1} is `False`. In turn this implies that u_{n-2} must also be `False`, and by extension all the u_i for $3 \leq i \leq n-2$. However in that case the first clause fails as u_3 is `False` and this means the conjunction of all the clauses evaluates as `False`.

In order to finalize the proof we need to ensure that the proposed transformations are applicable in polynomial time. This is clearly the case since the number of additional literals is polynomial in the number of original literals.

Hence $SAT \ltimes$ 3-SAT and 3-SAT is $\mathcal{NP}$-complete. $\square$

Key points

- How to measure the running time of an algorithm?
- How is the growth rate evolving?
- What is the inverse Ackerman's function?
- Recall the Master Theorem.
- How to relate Turing machines to algorithms?



Thank you!