Space Complexity of Algorithms

- So far we have considered only the time necessary for a computation
- Sometimes the size of the memory necessary for the computation is more critical.

The **amount of memory** used by machine $\mathcal{M}$ in a computation on input w can be for example:

- the maximal number of bits necessary for storing all data for each configuration
- the maximal number of memory cells used during the computation

Definition

A **space complexity** of algorithm Alg running on machine $\mathcal{M}$ is the function $S : \mathbb{N} \to \mathbb{N}$, where $S(n)$ is the maximal amount of memory used by $\mathcal{M}$ for inputs of size n.

Space Complexity of Algorithms

- There can be two algorithms for a particular problem such that one of them has a smaller time complexity and the other a smaller space complexity.

- If the time-complexity of an algorithm is in $O(f(n))$ then also the space complexity is in $O(f(n))$ (note that the number of memory cells used in one instruction is bounded by some constant that does not depend on the size of an input).

- The space complexity can be much smaller than the time complexity — the space complexity of INSERTION-SORT is $\Theta(n)$, while its time complexity is $\Theta(n^2)$.

Complexity of Algorithms

Some typical values of the size of an input n, for which an algorithm with the given time complexity usually computes the output on a “common PC” within a fraction of a second or at most in seconds.

(Of course, this depends on particular details. Moreover, it is assumed here that no big constants are hidden in the asymptotic notation)

$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
1 000 000 – 100 000 000	100 000 – 1 000 000	1000 – 10 000	100 – 1000

$2^{O(n)}$	$O(n!)$
20 – 30	10 – 15

Complexity of Algorithms

When we use asymptotic estimations of the complexity of algorithms, we should be aware of some issues:

- Asymptotic estimations describe only how the running time grows with the growing size of input instance.
- They do not say anything about exact running time. Some big constants can be hidden in the asymptotic notation.
- An algorithm with better asymptotic complexity than some other algorithm can be in reality faster only for very big inputs.
- We usually analyze the time complexity in the worst case. For some algorithms, the running time in the worst case can be much higher than the running time on "typical" instances.

Complexity of Algorithms

- This can be illustrated on algorithms for sorting.

Algorithm	Worst-case	Average-case
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$

- Quicksort has a worse asymptotic complexity in the worst case than Heapsort and the same asymptotic complexity in an average case but it is usually faster in practice.

Complexity of Algorithms

Polynomial — an expression of the form

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

where $a_0, a_1, \ldots, a_k$ are constants.

Examples of polynomials:

$$4n^3 - 2n^2 + 8n + 13 \qquad 2n + 1 \qquad n^{100}$$

Function f is called **polynomial** if it is bounded from above by some polynomial, i.e., if there exists a constant k such that $f \in O(n^k)$.

For example, the functions belonging to the following classes are polynomial:

$$O(n) \qquad O(n \log n) \qquad O(n^2) \qquad O(n^5) \qquad O(\sqrt{n}) \qquad O(n^{100})$$

Complexity of Algorithms

Function such as 2^n or $n!$ are not polynomial — for arbitrarily big constant k we have

$$2^n \in \Omega(n^k) \qquad\qquad n! \in \Omega(n^k)$$

Polynomial algorithm — an algorithm whose time complexity is polynomial (i.e., bounded from above by some polynomial)

Roughly we can say that:

- polynomial algorithms are effiecient algorithms that can be used in practice for inputs of considerable size
- algorithms, which are not polynomial, can be used in practice only for rather small inputs

Complexity of Algorithms

The division of algorithms on polynomial and non-polynomial is very rough — we cannot claim that polynomial algorithms are always efficient and non-polynomial algorithms are not:

- an algorithm with the time complexity $\Theta(n^{100})$ is probably not very useful in practice,
- some algorithms, which are non-polynomial, can still work very efficiently for majority of inputs, and can have a time complexity bigger than polynomial only due to some problematic inputs, on which the computation takes long time.

Remark: Polynomial algorithms where the constant in the exponent is some big number (e.g., algorithms with complexity $\Theta(n^{100})$) almost never occur in practice as solutions of usual algorithmic problems.

Complexity of Algorithms

For most of common algorithmic problems, one of the following three possibilities happens:

- A polynomial algorithm with time complexity $O(n^k)$ is known, where k is some very small number (e.g., 5 or more often 3 or less).
- No polynomial algorithm is known and the best known algorithms have complexities such as $2^{\Theta(n)}$, $\Theta(n!)$, or some even bigger.

  In some cases, a proof is known that there does not exist a polynomial algorithm for the given problem (it cannot be constructed).
- No algorithm solving the given problem is known (and it is possibly proved that there does not exist such algorithm)

Complexity of Algorithms

A typical example of polynomial algorithm — matrix multiplication with time complexity $\Theta(n^3)$ and space complexity $\Theta(n^2)$:

Algorithm 1: Matrix multiplication

```
MATRIX-MULT $(A, B, C, n)$:
    for $i := 1$ to $n$ do
        for $j := 1$ to $n$ do
            $x := 0$
            for $k := 1$ to $n$ do
                $x := x + A[i][k] * B[k][j]$
            $C[i][j] := x$
```

Complexity of Algorithms

- For a rough estimation of complexity, it is often sufficient to count the number of nested loops — this number then gives the degree of the polynomial

  Example: Three nested loops in the matrix multiplication — the time complexity of the algorithm is $O(n^3)$.

- If it is not the case that all the loops go from 0 to n but the number of iterations of inner loops are different for different iterations of an outer loops, a more precise analysis can be more complicated.

  It is often the case, that the sum of some sequence (e.g., the sum of arithmetic or geometric progression) is then computed in the analysis.

  The results of such more detailed analysis often does not differ from the results of a rough analysis but in many cases the time complexity resulting from a more detailed analysis can be considerably smaller than the time complexity following from the rough analysis.

Complexity of Algorithms

Arithmetic progression — a sequence of numbers $a_0, a_1, \ldots, a_{n-1}$, where

$$a_i = a_0 + i \cdot d,$$

where d is some constant independent on i.

Remark: So in an arithmetic progression, we have $a_{i+1} = a_i + d$ for each i.

The sum of an arithmetic progression:

$$\sum_{i=0}^{n-1} a_i \;=\; a_0 + a_1 + \cdots + a_{n-1} \;=\; \frac{1}{2} n \left(a_{n-1} + a_0\right)$$

Complexity of Algorithms

Example:

$$1 + 2 + \cdots + n \;=\; \frac{1}{2}n(n+1) \;=\; \frac{1}{2}n^2 + \frac{1}{2}n \;=\; \Theta(n^2)$$

For example, for $n = 100$ we have

$$1 + 2 + \cdots + 100 \;=\; 50 \cdot 101 \;=\; 5050.$$

Remark: To see this, we can note that

$$1 + 2 + \cdots + 100 \;=\; (1 + 100) + (2 + 99) + \cdots + (50 + 51),$$

where we compute the sum of 50 pairs of number, where the sum of each pair is 101.

Complexity of Algorithms

Geometric progression — a sequence of numbers $a_0, a_1, \ldots, a_n$, where

$$a_i = a_0 \cdot q^i,$$

where q is some constant independent on i.

Remark: So in a geometric progression we have $a_{i+1} = a_i \cdot q$.

The sum of a geometic progression (where $q \neq 1$):

$$\sum_{i=0}^{n} a_i = a_0 + a_1 + \cdots + a_n \;=\; a_0 \frac{q^{n+1} - 1}{q - 1}$$

Complexity of Algorithms

Example:

$$1 + q + q^2 + \cdots + q^n \;=\; \frac{q^{n+1} - 1}{q - 1}$$

In particular, for $q = 2$:

$$1 + 2^1 + 2^2 + 2^3 + \cdots + 2^n \;=\; \frac{2^{n+1} - 1}{2 - 1} \;=\; 2 \cdot 2^n - 1 \;=\; \Theta(2^n)$$

Complexity of Algorithms

An **exponential** function: a function of the form c^n, where c is a constant — e.g., function 2^n

Logarithm — the inverse function to an exponential function: for a given n,

$$\log_c n$$

is the value x such that $c^x = n$.

Complexity of Algorithms

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	131072
18	262144
19	524288
20	1048576

n	$\lceil \log_2 n \rceil$
0	—
1	0
2	1
3	2
4	2
5	3
6	3
7	3
8	3
9	4
10	4
11	4
12	4
13	4
14	4
15	4
16	4
17	5
18	5
19	5
20	5

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20

Complexity of Algorithms

Proposition

For any $a, b > 1$ and any $n > 0$ we have

$$\log_a n = \frac{\log_b n}{\log_b a}$$

Proof: From $n = a^{\log_a n}$ it follows that $\log_b n = \log_b(a^{\log_a n})$.
Since $\log_b(a^{\log_a n}) = \log_a n \cdot \log_b a$, we obtain $\log_b n = \log_a n \cdot \log_b a$, from which the above mentioned conclusion follows directly. $\square$

Due to this observation, the base of a logarithm is often omited in the asymptotic notation: for example, instead of $\Theta(n \log_2 n)$ we can write $\Theta(n \log n)$.

Complexity of Algorithms

Examples where exponential functions and logarithms can appear in an analysis of algorithms:

- Some value is repeatedly decreased to one half or is repeatedly doubled.

  For example, in the **binary search**, the size of an interval halves in every iteration of the loop.

  Let us assume that an array has size n.

  What is the minimal size of an array n, for which the algorithm performs at least k iterations?

  The answer: 2^k

  So we have $k = \log_2(n)$. The time complexity of the algorithm is then $\Theta(\log n)$.

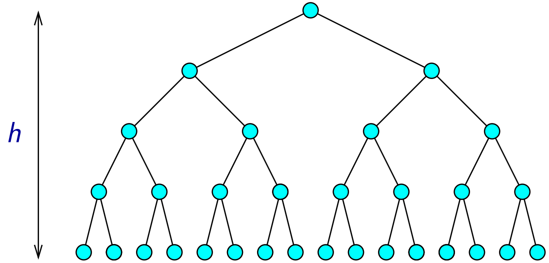
Complexity of Algorithms

- Using n bits we can represent numbers from 0 to $2^n - 1$.
- The minimal numbers of bits, which are sufficient for representing a natural number x in binary is
$$\lceil \log_2(x+1) \rceil .$$
- A perfectly balanced tree of height h has $2^{h+1} - 1$ nodes, and 2^h of these nodes are leaves.
- The height of a perfectly balanced binary tree with n nodes is $\log_2 n$.

  An illustrating example: If we would draw a balanced tree with $n = 1\,000\,000$ nodes in such a way that the distance between neighbouring nodes would be 1 cm and the height of each layer of nodes would be also 1 cm, the width of the tree would be 10 km and its height would be approximately 20 cm.

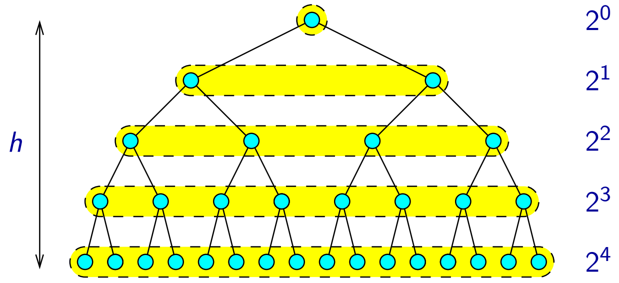
Complexity of Algorithms

A perfectly balanced binary tree of height h:



Complexity of Algorithms

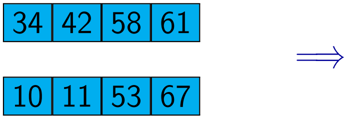
A perfectly balanced binary tree of height h:



Complexity of Algorithms

Example: Algorithm MERGE-SORT.

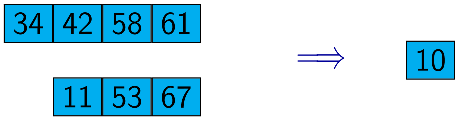
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Complexity of Algorithms

Example: Algorithm MERGE-SORT.

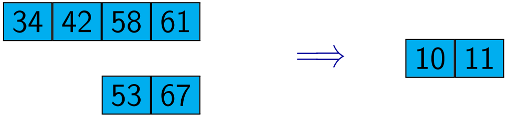
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Complexity of Algorithms

Example: Algorithm MERGE-SORT.

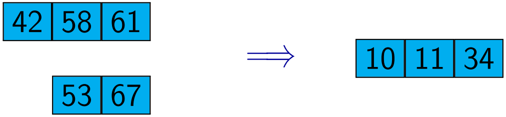
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Complexity of Algorithms

Example: Algorithm MERGE-SORT.

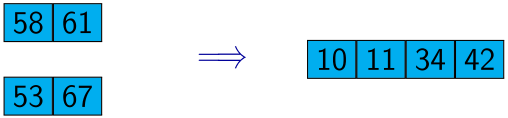
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Complexity of Algorithms

Example: Algorithm MERGE-SORT.

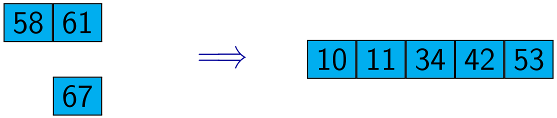
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Complexity of Algorithms

Example: Algorithm MERGE-SORT.

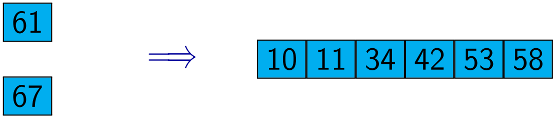
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Complexity of Algorithms

Example: Algorithm Merge-Sort.

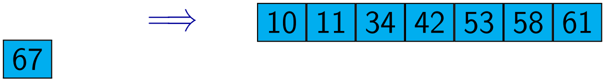
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Complexity of Algorithms

Example: Algorithm MERGE-SORT.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Complexity of Algorithms

Example: Algorithm MERGE-SORT.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.

$\Longrightarrow$

10	11	34	42	53	58	61	67

Complexity of Algorithms

Algorithm 2: Merge sort

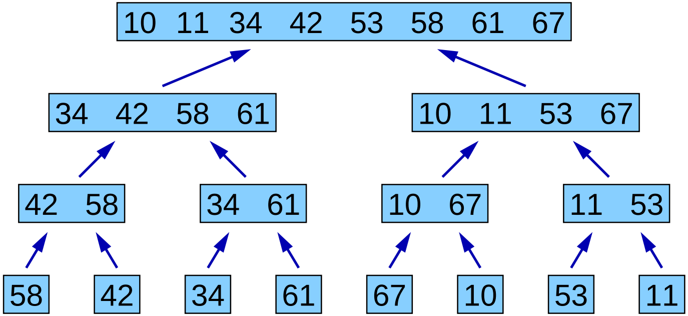
```
MERGE-SORT $(A, p, r)$:
    if $r - p > 1$ then
        $q := \lfloor (p + r) / 2 \rfloor$
        MERGE-SORT$(A, p, q)$
        MERGE-SORT$(A, q, r)$
        MERGE$(A, p, q, r)$
```

To sort an array A containing elements $A[0], A[1], \cdots, A[n-1]$ we call $\textsc{Merge-Sort}(A, 0, n)$.

Remark: Procedure $\textsc{Merge}(A, p, q, r)$ merges sorted sequences stored in $A[p \mathinner{..} q-1]$ and $A[q \mathinner{..} r-1]$ into one sequence stored in $A[p \mathinner{..} r-1]$.

Complexity of Algorithms

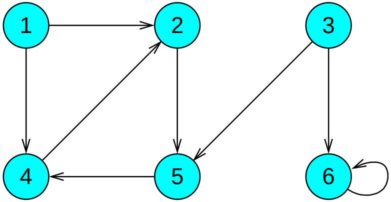
Input: 58, 42, 34, 61, 67, 10, 53, 11

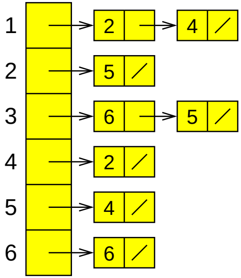


The tree of recursive calls has $\Theta(\log n)$ layers. On each layer, $\Theta(n)$ operations are performed. The time complexity of MERGE-SORT is $\Theta(n \log n)$.

Complexity of Algorithms

Representations of a graph:

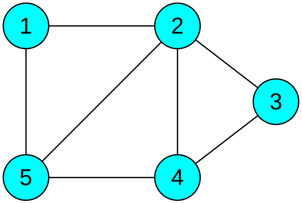


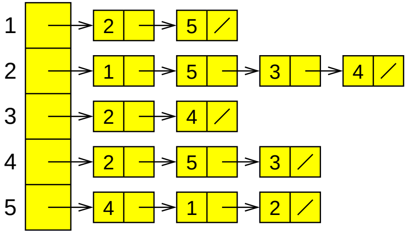


$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

Complexity of Algorithms

Representations of a graph:





$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

Complexity of Algorithms

Finding the shortest path in a graph where edges are not weighted:

- Breadth-first search
- The input is a graph G (with a set of nodes V) and an initial node s.
- The algorithm finds the shortest paths from node s for all nodes.
- For a graph with n nodes and m edges, the running time of the algorithm is $\Theta(n + m)$.

Complexity of Algorithms

Algorithm 3: Breadth-first search

```
BFS $(G, s)$:
    BFS-INIT$(G, s)$
    ENQUEUE$(\mathcal{Q}, s)$
    while $\mathcal{Q} \neq \emptyset$ do
        $u :=$ DEQUEUE$(\mathcal{Q})$
        for each $v \in edges[u]$ do
            if $color[v] = $ WHITE then
                $color[v] :=$ GRAY
                $d[v] := d[u] + 1$
                $pred[v] := u$
                ENQUEUE$(\mathcal{Q}, v)$
        $color[u] :=$ BLACK
```

Complexity of Algorithms

Algorithm 4: Breadth-first search — initialization

```
$\textsc{Bfs-Init}\,(G, s)$:
    for each $u \in V - \{s\}$ do
        $color[u] := \textsc{white}$
        $d[u] := \infty$
        $pred[u] := \textsc{nil}$
    $color[s] := \textsc{gray}$
    $d[s] := 0$
    $pred[s] := \textsc{nil}$
    $Q := \emptyset$
```

Other Examples of Problems

Problem "Primality"

Input: A natural number x.

Output: YES if x is a prime, NO otherwise.

Remark: A natural number x is a **prime** if it is greater than 1 and is divisible only by numbers 1 and x.

Few of the first primes: $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, \ldots$

Decision Problems

The problems, where the set of outputs is $\{\textsc{Yes}, \textsc{No}\}$ are called **decision problems**.

Decision problems are usually specified in such a way that instead of describing what the output is, a question is formulated.

Example:

Problem "Primality"

Input:	A natural number x.
Question:	Is x a prime?

Primality Test

A simple algorithm solvint the “Primality” problem can work like this:

- Test the trivial cases (e.g., if $x \leq 2$ or if x is even).
- Try to divide x successively by all odd numbers in interval $3, \ldots, \lfloor\sqrt{x}\rfloor$.

Let n be the number of bits in the representation of number x, e.g., $n = \lceil \log_2(x+1) \rceil$.
This value n will be considered as the size of the input.

Note that the number $\lfloor\sqrt{x}\rfloor$ has approximately $n/2$ bits.

There are approximately $2^{(n/2)-1}$ odd numbers in the interval $3, \ldots, \lfloor\sqrt{x}\rfloor$, and so the time complexity of this simple algorithm is in $2^{\Theta(n)}$.

Primality Test

This simple algorithm with an exponential running time (resp. also different improved versions of this) are applicable to numbers with thousands of bits in practice.
A primality test of such big numbers plays an important role for example in **cryptography**.

Only since 2003, a polynomial time algorithm is known. The time complexity of the original version of the algorithm was $O(n^{12+\varepsilon})$, later it was improved to ($O(n^{7.5})$). The currently fastest algorithm has time complexity $O(n^6)$.

In practice, **randomized algorithms** are used for primality testing:

- Solovay–Strassen
- Miller–Rabin

(The time complexity of both algorithms is $O(n^3)$.)

Primality Test

A **randomized algorithm**:

- It uses a random-number generator during a computation.
- It can produce different outputs in different runs with the same input.
- The output need not be always correct but the probability of producing an incorrect output is bounded.

For example, both above mentioned randomized algorithms for primality testing behave as follows:

- If x is a prime, the answer YES is always returned.
- If x is not a prime, the probability of the answer NO is at least 50% but there is at most 50% probability that the program returns the incorrect answer YES.

Primality Test

The program can be run repeatedly (k times):

- If the program returns at least once the answer $\textsc{No}$, we know (with 100% probability) that x is not a prime.
- If the program always returns $\textsc{Yes}$, the probability that x is not a prime is at most $\frac{1}{2^k}$.

For sufficiently large values of k, the probability of an incorrect answer is negligible.

Remark: For example for $k = 100$, the probability of this error is smaller than the probability that a computer, on which the program is running, will be destroyed by a falling meteorite (assuming that at least once in every 1000 years at least $100\,\mathrm{m}^2$ of Earth surface is destroyed by a meteorite).

Primality Test

At first sight, the following problem looks very similar as the primality test:

Problem "Factorization"

Input: A natural number x, where $x > 1$.

Output: Primes $p_1, p_2, \ldots, p_m$ such that $x = p_1 \cdot p_2 \cdot \cdots \cdot p_m$.

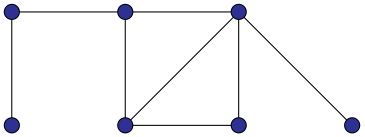
In fact, this problem is (supposed to be) much harder than primality testing.
No efficient (polynomial) algorithm is known for this problem (nor a randomized algorithm).

Other Examples of Problems

Independent set (IS) problem

Input: An undirected graph G, a number k.

Question: Is there an independent set of size k in the graph G?



$k = 4$

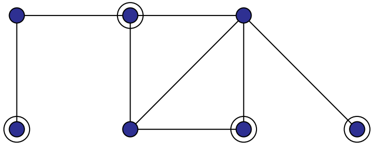
Remark: An **independent set** in a graph is a subset of nodes of the graph such that no pair of nodes from this set is connected by an edge.

Other Examples of Problems

Independent set (IS) problem

Input: An undirected graph G, a number k.

Question: Is there an independent set of size k in the graph G?

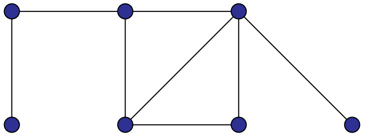


$k = 4$

Remark: An **independent set** in a graph is a subset of nodes of the graph such that no pair of nodes from this set is connected by an edge.

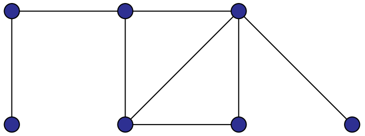
Other Examples of Problems

An example of an instance where the answer is YES:



$k = 4$

An example of an instance where the answer is NO:



$k = 5$

Other Examples of Problems

- A set containing n elements has 2^n subsets.

  Consider for example an algorithm solving a given problem by **brute force** where it tests the required property for each subset of a given set.

- It is sufficient to consider only subsets of size k. The total number of such subsets is
  $$\binom{n}{k}$$
  For some values of k, the total number of these subsets is not much smaller than 2^n:

  For example, it is not too difficult to show that
  $$\binom{n}{\lfloor n/2 \rfloor} \geq \frac{2^n}{n}.$$

Other Examples of Problems

Let us have an algorithm solving the independent set problem by brute force in such a way that it tests for each subset with k elements of the set of nodes (with n nodes), if it forms an independent set. The time complexity of the algorithm is $2^{\Theta(n)}$.