NP-ПЪЛНИ ЗАДАЧИ

Алгоритмичните задачи се делят на различни видове според върнатата стойност (резултата): задачи за разпознаване, оптимизационни задачи, комбинаторни задачи и др.

Задачите за разпознаване връщат един бит, който може да се тълкува като *да* или *не* — отговор на общ въпрос. Тоест тяхната върната стойност е от логически тип: истина или лъжа. В програмен код ще представяме отговора *да* като истина, а *не* — като лъжа.

В оптимизационните задачи се търси най-голяма или най-малка стойност. Те се свеждат до задачи за разпознаване чрез замяна на бившия изход с допълнителен входен параметър L: "Достижима ли е стойност, по-голяма (по-малка) от L?"

В комбинаторните задачи се търси броят на някакви обекти. И този тип задачи се свеждат до задачи за разпознаване: "Съществуват ли поне L обекта от съответния вид?"

Тези превръщания на типовете задачи ни дават основание да отредим централно място на задачите за разпознаване. Оттук нататък ще разглеждаме само такива задачи.

Времевата сложност на алгоритъм разглеждаме като функция на дължината n на входа. Казваме, че сложността е полиномиална, ако тя е ограничена отгоре от степенна функция на n с цял неотрицателен показател (неравенството е в асимптотичен смисъл, тоест при $n \to \infty$). Например изразите $n^7 \log n$ и $n^7\sqrt{n}$, разглеждани просто като функции на n, не са полиноми, обаче са полиномиални сложности, защото са ограничени отгоре от n^8.

Тъй като може да има различни входни данни с една и съща дължина n, нека уточним, че се интересуваме от сложността при най-лоши входни данни. Ако става дума за оценка отгоре, тя се отнася еднакво за най-лоши входни данни и за всички входни данни.

Със знака **P** е прието да се означава класът на алгоритмичните задачи за разпознаване, за които съществува алгоритъм с полиномиална времева сложност при най-лоши входни данни. Понеже полиномиалните сложности са най-малките, то задачите от класа **P** се смятат за лесни. Това невинаги е така в действителност: ако времето за обработка е полином от висока степен, то може да се окаже твърде голямо дори за не много дълги входни данни. Но в повечето случаи тази преценка е смислена: задачите от **P** се приемат за лесни, а задачите извън **P** — за трудни.

Неформално казано, **NP** е класът на алгоритмичните задачи за разпознаване, за които има бърз алгоритъм за проверка на предложено решение. "Бърз" значи "с полиномиална сложност". Разликата между решаване на задача за разпознаване и проверка на предложено решение може да се онагледи с въпросите: "Уравнението $a_0 + a_1x + \ldots + a_nx^n = 0$ има ли реален корен?" и "Реалното число x_0 корен ли е на уравнението $a_0 + a_1x + \ldots + a_nx^n = 0$?" Вторият въпрос съдържа един входен параметър повече — предложеното решение x_0.

Допълнителният параметър официално се нарича сертификат. Той трябва да бъде къс (дължината му не бива да надхвърля полином на дължината на останалата част от входа), иначе никой алгоритъм не би могъл дори да прочете, камо ли да провери сертификата бързо.

Съществуването на сертификат е задължително само при отговор *да*. Когато например едно алгебрично уравнение има реален корен, той може да бъде предложен като сертификат. Но ако уравнението няма реален корен, не е ясно какъв сертификат можем да предложим в потвърждение на този факт.

Сега можем да съставим точно определение: **NP** е класът на задачите за разпознаване, за които съществува алгоритъм с полиномиална времева сложност при всякакви входни данни, проверяващ отговора *да* на задачата с помощта на допълнителен параметър, наречен сертификат, който зависи от входните данни на задачата и чиято дължина е полиномиална спрямо тяхната.

Задачите за разпознаване, притежаващи къс сертификат и бърз (полиномиален) алгоритъм за проверка на отговор *не*, образуват класа **co-NP**.

Точното поставяне на въпросите на алгоритмичните задачи за разпознаване е важно поради асиметрията между отговорите *да* и *не*.

Очевидно е, че ако има бърз алгоритъм, който решава задачата, без да използва сертификат, то същият алгоритъм може да се използва и за проверка на отговора с помощта на сертификат: алгоритъмът просто пренебрегва сертификата. Затова $\mathbf{P} \subseteq \mathbf{NP}$ и $\mathbf{P} \subseteq \mathbf{co\text{-}NP}$. Предполага се, че двете включвания са строги, но засега няма доказателство за това предположение.

Нека $\mathcal{A}$ и $\mathcal{B}$ са алгоритмични задачи. Полиномиална редукция от задачата $\mathcal{A}$ към задачата $\mathcal{B}$ наричаме алгоритъм от следния вид:

$\mathcal{A}\big($inputA: входни данни на задачата $\mathcal{A}\big)$
1) inputB $\leftarrow$ Редукция на входа$\big($inputA$\big)$
2) outputB $\leftarrow$ $\mathcal{B}\big($inputB$\big)$
3) outputA $\leftarrow$ Редукция на изхода$\big($outputB$\big)$
4) **return** outputA

Изисква се редукцията на входа и редукцията на изхода да имат полиномиална времева сложност по отношение на дължината на входа на задачата $\mathcal{A}$. Едната от двете редукции може да липсва; най-често това е редукцията на изхода:

$\mathcal{A}\big($inputA: входни данни на задачата $\mathcal{A}\big)$
1) inputB $\leftarrow$ Редукция на входа$\big($inputA$\big)$
2) **return** $\mathcal{B}\big($inputB$\big)$

Обратно, определението може да се разшири, като се допусне многократно извикване (в цикъл) на задачата $\mathcal{B}$; обаче броят на извикванията трябва да е полином от дължината на входа на $\mathcal{A}$.

Наличието на полиномиална редукция от $\mathcal{A}$ към $\mathcal{B}$ се бележи така: $\mathcal{A} \propto_{\mathrm{P}} \mathcal{B}$.

Независимо от вида на полиномиалната редукция, наличието ѝ показва, че ако задачата $\mathcal{B}$ може да се реши за полиномиално време, то и задачата $\mathcal{A}$ е решима за полиномиално време. С помощта на контрапозиция следва, че ако задачата $\mathcal{A}$ е нерешима за полиномиално време, то и задачата $\mathcal{B}$ не може да се реши за полиномиално време. Нещо повече, в последния случай времевата сложност на задачата $\mathcal{B}$ е горна граница за времевата сложност на задачата $\mathcal{A}$. Ако задачата $\mathcal{B}$ може да се реши например за време 3^n, то редукцията произвежда алгоритъм, който решава задачата $\mathcal{A}$ за време $3^n + \text{полином}(n) = \Theta(3^n)$. Нищо не пречи да съществува и по-бърз алгоритъм за задачата $\mathcal{A}$ — такъв, който не се обръща към задачата $\mathcal{B}$. Ето защо за задачи извън класа **P** можем неформално да четем обозначението $\mathcal{A} \propto_{\mathrm{P}} \mathcal{B}$ по следния начин: "Задачата $\mathcal{A}$ е не по-трудна от задачата $\mathcal{B}$."

Редукцията често се използва за долни граници, тоест $\mathcal{A}$ е известна, а $\mathcal{B}$ — нова задача. Тогава правим извод (за задачи извън **P**), че задачата $\mathcal{B}$ е не по-лесна от $\mathcal{A}$.

Алгоритмичната задача за разпознаване $\mathcal{B}$ се нарича **NP**-трудна, ако $\forall \mathcal{A} \in \mathbf{NP} : \mathcal{A} \propto_{\mathrm{P}} \mathcal{B}$.

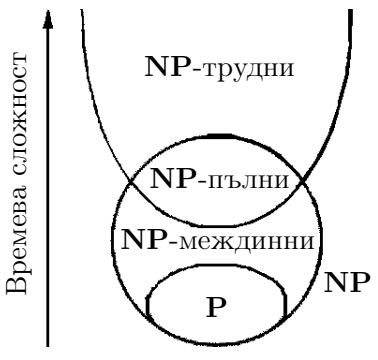
Това определение е удобно за теорията, но не и за практиката: ако се стремим да докажем, че някоя задача е **NP**-трудна, не е лесно да сведем до нея всяка задача от **NP**.

Теорема: Ако $\mathcal{C}$ е **NP**-трудна задача и $\mathcal{C} \propto_{\mathrm{P}} \mathcal{D}$, то $\mathcal{D}$ също е **NP**-трудна задача.

Доказателство: Щом $\mathcal{C}$ е **NP**-трудна задача, то $\forall \mathcal{A} \in \mathbf{NP} : \mathcal{A} \propto_{\mathrm{P}} \mathcal{C}$. По условие $\mathcal{C} \propto_{\mathrm{P}} \mathcal{D}$. Следователно $\forall \mathcal{A} \in \mathbf{NP} : \mathcal{A} \propto_{\mathrm{P}} \mathcal{D}$. Затова $\mathcal{D}$ е **NP**-трудна задача.

Тази теорема е по-удобна от определението: достатъчна е една редукция, а не безброй. За практическото прилагане на теоремата е нужно да знаем достатъчно много **NP**-трудни задачи.

Една задача за разпознаване се нарича **NP**-пълна, ако е **NP**-трудна и принадлежи на **NP**. С други думи, **NP**-пълните задачи са най-трудните задачи в класа **NP**. Обратно, задачите от **P** са най-лесните задачи в **NP**. Ако $\mathbf{P} \neq \mathbf{NP}$, то никоя задача от **P** не е **NP**-пълна и двете множества заедно не изчерпват класа **NP**, а остават т. нар. **NP**-междинни задачи, за които няма алгоритъм с полиномиална времева сложност, но които въпреки това не са **NP**-пълни. Понеже не е доказано, че $\mathbf{P} \neq \mathbf{NP}$, то за никоя задача от **NP** не е сигурно, че е **NP**-междинна.



ВАЖНИ АЛГОРИТМИЧНИ ЗАДАЧИ ОТ NP

1) SAT (удовлетворимост на логическа формула; от англ. satisfiability):
— Вход: конюнктивна нормална форма F.
— Въпрос: F удовлетворима ли е?

Теорема на Кук—Левин: SAT е **NP**-пълна задача.

2) 2-SAT е частен случай на SAT: всички клаузи са дизюнкции с два операнда.

Пример: $(\overline{x} \vee y) \wedge (t \vee x) \wedge (y \vee t)$.

Задачата 2-SAT $\in \mathbf{P}$, тоест за нея съществува бърз (полиномиален) алгоритъм;
основава се на търсене на компонентите на силна свързаност на подходящ граф.

3) 3-SAT е частен случай на SAT: всички клаузи са дизюнкции с три операнда.

Пример: $(\overline{x} \vee y \vee t) \wedge (t \vee x \vee \overline{y})$.

Задачата 3-SAT е **NP**-пълна.

Частният случай не може да бъде по-труден от общия. При задачата SAT се осъществяват и двете възможности: 2-SAT е по-лесна от SAT, а 3-SAT е със същата трудност като SAT.

4) SubsetSum (търсене на подмножество с даден сбор) е **NP**-пълна задача:
— Вход: $A\big[1 \ldots n\big]$ — цели положителни числа; S — цяло неотрицателно число.
— Въпрос: S може ли да се представи като сбор от елементи на $A\big[1 \ldots n\big]$?
По-формално: $\exists$ ли $M \subseteq \big\{1\,;\,2\,;\,\ldots\,;\,n\big\} : \sum\limits_{k \,\in\, M} A\big[k\big] = S$?

5) Partition (разбиване на множество на две части с равни сборове) е **NP**-пълна задача:
— Вход: $A\big[1 \ldots n\big]$ — цели положителни числа.
— Въпрос: $A\big[1 \ldots n\big]$ може ли да се разбие на две части с равни сборове?
По-формално: $\exists$ ли $M \subseteq \big\{1\,;\,2\,;\,\ldots\,;\,n\big\} : \sum\limits_{k \,\in\, M} A\big[k\big] = \sum\limits_{k \,\notin\, M} A\big[k\big]$?

6) Задачата за раницата е **NP**-пълна:
— Вход: $V\big[1 \ldots n\big]$ и $W\big[1 \ldots n\big]$ — цели положителни числа; C и L — цели неотрицателни числа.
— Въпрос: $\exists$ ли $M \subseteq \big\{1\,;\,2\,;\,\ldots\,;\,n\big\} : \sum\limits_{k \,\in\, M} W\big[k\big] \leq C$ и $\sum\limits_{k \,\in\, M} V\big[k\big] \geq L$?

7) “Изоморфен подграф” е **NP**-пълна задача:
— Вход: два графа G и H.
— Въпрос: G съдържа ли подграф, изоморфен на H?

8) “Изоморфни графи” вероятно е **NP**-междинна задача:
— Вход: два графа G и H.
— Въпрос: Изоморфни ли са G и H?

9) “Максимално съчетание” е задача от **P**:
— Вход: граф G и цяло неотрицателно число L.
— Въпрос: G съдържа ли съчетание с поне L ребра?
Съчетание (сдвояване) се нарича множество от ребра, никои две от които нямат общ край.

10) "Върхово покритие" е **NP**-пълна задача:
— Вход: граф G и цяло неотрицателно число L.
— Въпрос: G притежава ли върхово покритие с не повече от L върха?
Върхово покритие е множество от върхове, съдържащо поне един край на всяко ребро.

11) "Доминиращо множество" е **NP**-пълна задача:
— Вход: граф G и цяло неотрицателно число L.
— Въпрос: G притежава ли доминиращо множество с не повече от L върха?
Доминиращо е такова множество от върхове, че всеки друг връх е съседен на някой от тях.

12) "Клика" е **NP**-пълна задача:
— Вход: граф G и цяло неотрицателно число K.
— Въпрос: G съдържа ли клика с поне (точно) K върха?
Клика се нарича всеки пълен подграф.

13) "Антиклика" ("Независимо множество") е **NP**-пълна задача:
— Вход: граф G и цяло неотрицателно число K.
— Въпрос: G съдържа ли антиклика с поне (точно) K върха?
Антиклика се нарича всеки празен подграф.

14) Задачите "Ойлеров път" и "Ойлеров цикъл" са от класа **P**:
— Вход: ориентиран или неориентиран граф G.
— Въпрос: G съдържа ли ойлеров път (ойлеров цикъл)?
Един път (цикъл) се нарича ойлеров, ако минава точно веднъж по всяко ребро на графа.

15) "Хамилтонов път" е **NP**-пълна задача:
— Вход: ориентиран или неориентиран граф G и нула, един или два върха s и t на G.
— Въпрос: G съдържа ли хамилтонов път (от s)(до t)?
Един път се нарича хамилтонов, ако посещава точно веднъж всеки връх на графа.
NP-пълни са всички варианти — със или без зададен начален и краен връх.

16) "Хамилтонов цикъл" е **NP**-пълна задача:
— Вход: ориентиран или неориентиран граф G.
— Въпрос: G съдържа ли хамилтонов цикъл?
Един цикъл се нарича хамилтонов, ако посещава точно веднъж всеки връх на графа.

17) Задачата за търговския пътник е **NP**-пълна:
— Вход: тегловен граф G и реално число L.
Обикновено графът G е пълен, а числото L и теглата на ребрата са положителни.
— Въпрос: G съдържа ли хамилтонов цикъл с дължина, ненадвишаваща L?

18) "Най-дълъг прост път" е **NP**-пълна задача:
— Вход: ориентиран или неориентиран граф G (тегловен или нетегловен);
нула, един или два върха s и t на G; реално число L.
Обикновено числото L и теглата на ребрата са положителни.
— Въпрос: G съдържа ли прост път (от s)(до t) с дължина поне L?
Един път се нарича прост, ако не повтаря върхове.

Известни са стотици задачи от класа **NP**, много от които имат важни практически приложения. Тук се ограничаваме с приложения списък.

ЗАДАЧИ ЗА УПРАЖНЕНИЕ

Задача 1. Да се докаже, че е **NP**-пълна алгоритмичната задача Problem1:
— Вход: $A[1 \ldots n]$ — цели положителни числа; S и k — цели неотрицателни числа.
— Въпрос: S може ли да се представи като сбор от k-ти степени на елементи на $A[1 \ldots n]$?
По-формално: $\exists$ ли $M \subseteq \{1\,;\, 2\,;\, \ldots\,;\, n\} : \sum\limits_{i \in M} \big(A[i]\big)^k = S$?

Решение: Очевидна е приликата между Problem1 и SubsetSum. Обаче трябва да внимаваме за посоката на редукцията. Едно решение в грешна посока изглежда така:

```
Problem1(A[1...n], S, k)
  1) for i ← 1 to n do
  2)       A[i] ← (A[i])^k
  3) return SubsetSum(A[1...n], S)
```

По-точно, самият алгоритъм е верен, но не ни казва нищо за трудността на задачата Problem1. Следва само, че задачата Problem1 е не по-трудна от SubsetSum. Оттук не можем да разберем дали Problem1 е **NP**-пълна задача, или принадлежи на **P**. Може би предложеният алгоритъм е просто бавен метод за решаването на задачата Problem1, който обаче не изключва наличието на бърз (полиномиален) алгоритъм — такъв, който не се обръща към задачата SubsetSum.

Трябва да съставим полиномиална редукция в обратната посока: SubsetSum $\propto_{\mathrm{P}}$ Problem1. Неформално казано, задачата SubsetSum е частен случай на задачата Problem1 при $k = 1$. Формалното описание на редукцията изглежда така:

```
SubsetSum(A[1...n], S)
  1) k ← 1
  2) return Problem1(A[1...n], S, k)
```

Бързина на редукцията: Редукцията се състои в присвояването, извършвано от ред № 1. Това присвояване изразходва константно време: $\Theta(1)$, което е полином от нулева степен. Тоест изпълнено е изискването редукцията да бъде полиномиална.

Коректност на редукцията: Извършваме редица от преобразувания, с цел да проверим дали редукцията спазва изискванията на задачата SubsetSum.

Конкретният алгоритъм SubsetSum$\big(A[1 \ldots n],\, S\big)$ връща стойност "истина".

$\Updownarrow$ (от ред № 2 на алгоритъма)

Problem1$\big(A[1 \ldots n],\, S,\, k\big)$ връща стойност "истина".

$\Updownarrow$ (от постановката на задачата Problem1)

$$\exists M \subseteq \{1\,;\, 2\,;\, \ldots\,;\, n\} : \sum_{i \in M} \big(A[i]\big)^k = S.$$

$\Updownarrow$ ($k = 1$ според ред № 1 на алгоритъма)

$$\exists M \subseteq \{1\,;\, 2\,;\, \ldots\,;\, n\} : \sum_{i \in M} A[i] = S.$$

Като изоставим междинните резултати, получаваме следната еквивалентност:
Алгоритъмът SubsetSum$\big(A[1 \ldots n],\, S\big)$ връща "истина" $\iff \exists M \subseteq \{1\,;\, 2\,;\, \ldots\,;\, n\} : \sum\limits_{i \in M} A[i] = S.$
Това съвпада с постановката на задачата SubsetSum, следователно редукцията е коректна.

Дотук построихме полиномиална редукция SubsetSum $\propto_{\mathrm{P}}$ Problem1. От теорията знаем, че SubsetSum е **NP**-пълна, следователно **NP**-трудна задача. Въз основа на съответната теорема заключаваме, че Problem1 също е **NP**-трудна задача.

Доказателството на последното твърдение не се нуждае от нищо повече, но въпреки това ще приведем едно интуитивно разсъждение, с което да осветлим механизма на редукцията: Допускаме, че има бърз алгоритъм за Problem1. Добавяме бързата редукция (ред № 1 от кода) и получаваме бърз алгоритъм за SubsetSum. Но според теорията няма такъв (освен ако $\mathbf{P} = \mathbf{NP}$). Полученото противоречие показва, че направеното допускане не е било вярно. Вярно е обратното: не съществува бърз алгоритъм за задачата Problem1.

Разсъждението от предишния абзац е неточно, защото в него се предполага, че $\mathbf{P} \neq \mathbf{NP}$, и се пренебрегва класът на **NP**-междинните задачи. Въпреки това то е полезно, защото показва как работи редукцията.

Дотук доказахме, че Problem1 е **NP**-трудна задача. Остава да докажем, че тя е от **NP**. За целта трябва да предложим къс сертификат и бърз алгоритъм за проверка на отговор *да*. Понеже въпросът се отнася за съществуване, най-удобно е да изберем за сертификат това, за чието съществуване се пита — множеството M от индексите на елементите на числовия масив, избрани за образуване на сбора. Добре е да представим множеството M като логически масив: така избягваме повторенията на индекси и гарантираме, че всички индекси са допустими.

```
CheckProblem1 (A[1...n], S, k, M[1...n])
1)  sum ← 0
2)  for i ← 1 to n do
3)      if M[i]
4)          sum ← sum + (A[i])^k  // Бързо степенуване!
5)  return sum = S
```

Сертификатът $M[1 \ldots n]$ е къс, тъй като дължината му $\Theta(n)$ е полином (от първа степен) на дължината $\Theta(n + \log S + \log k)$ на останалата част от входа. Всяко изпълнение на ред № 4 изразходва време $\Theta(\log k)$, ако се използва бързото степенуване. Ред № 4 се изпълнява n пъти в най-лошия случай — когато всички елементи на масива $M[1 \ldots n]$ имат стойност "истина". Следователно времевата сложност на алгоритъма за проверка в най-лошия случай е $\Theta(n \log k)$, а това е полином (от втора степен) на дължината $\Theta(n + \log S + \log k)$ на входа без сертификата. Тъй като съществува къс сертификат и бърз алгоритъм за проверка, то задачата Problem1 $\in$ **NP**.

Щом Problem1 е **NP**-трудна задача и принадлежи на **NP**, то тя е **NP**-пълна.

Забележка: Ако ред № 4 от CheckProblem1 използва обикновено вместо бързо степенуване, то времевата сложност на CheckProblem1 в най-лошия случай ще притежава порядък $\Theta(nk)$. Тази сложност е псевдополиномиална (всъщност експоненциална), защото k е входна стойност, а не дължина на входа. Дължината (тоест броят на цифрите) на цялото число k е $\Theta(\log k)$. Тази реализация на CheckProblem1 е бавна: с нея не можем да докажем, че Problem1 $\in$ **NP**. Ето защо използването на бързото степенуване тук е задължително.

Задача 2. Да се докаже, че е **NP**-пълна алгоритмичната задача Problem2:
— Вход: неориентиран граф G и дърво T.
— Въпрос: G съдържа ли подграф, изоморфен на T?

Решение: Лесно се вижда, че Problem2 е частен случай на задачата "Изоморфен подграф", но това не доказва нищо: частният случай на **NP**-пълна задача може да е, но може и да не е **NP**-пълна задача. Обратно, в списъка с **NP**-пълни задачи трябва да потърсим такава задача, която да бъде частен случай на Problem2. За целта се налага да изберем подходящо дърво T, например прост път с n върха, където n е броят на върховете на графа G. Всеки път е дърво (т.е. допустима стойност за T), а щом съдържа n върха, той е хамилтонов път в G.

Описание на редукцията:

Хамилтонов път $(G$: неориентиран нетегловен граф с n върха$)$
1) $T \leftarrow (u_1 - u_2 - \cdots - u_n)$
2) **return** Problem2(G, T)

Бързина на редукцията: Редукцията се състои в построяването на дървото T на ред № 1. Това построение (n върха и $n-1$ ребра) отнема време $\Theta(n)$, което е полином от първа степен. Тоест изпълнено е изискването редукцията да бъде полиномиална.

Коректност на редукцията: От ред № 1 на кода следва, че T е прост път, а оттам и дърво. Следователно обръщението към задачата Problem2 на ред № 2 е с допустими входни данни. Остава да проверим дали редукцията спазва изискванията на задачата "Хамилтонов път".

Конкретният алгоритъм "Хамилтонов път" (G) връща стойност "истина".

$\Updownarrow$ (от ред № 2 на алгоритъма)

Problem2(G, T) връща стойност "истина".

$\Updownarrow$ (от постановката на задачата Problem2)

Графът G съдържа подграф, изоморфен на дървото T.

$\Updownarrow$ (T е прост път с n върха според ред № 1 на алгоритъма)

Графът G съдържа прост път с n върха, тоест хамилтонов път.

Като изоставим междинните резултати, получаваме следната еквивалентност:
Алгоритъмът "Хамилтонов път" (G) връща "истина" $\iff$ графът G съдържа хамилтонов път.
Това съвпада с постановката на задачата "Хамилтонов път", ето защо редукцията е коректна.

И така, има полиномиална редукция "Хамилтонов път" $\propto_{\mathrm{P}}$ Problem2. От теорията знаем, че "Хамилтонов път" е **NP**-пълна, следователно **NP**-трудна задача. От съответната теорема заключаваме, че Problem2 също е **NP**-трудна задача.

Нека графът G има n върха и m ребра, а дървото T има t върха. Тогава T има $t-1$ ребра. За сертификат ще използваме масив от цели числа $F[1 \ldots t]$, който определя съответствието между върховете на T и G: $F[i]$ е върхът на G, съответстващ на връх № i на T.

CheckProblem2(
G: неориентиран нетегловен граф с n върха и m ребра;
T: дърво с t върха и $t-1$ ребра;
$F[1 \ldots t]$: масив от цели числа // сертификат
)

```
 1) if t > n
 2)     return false // G не може да съдържа T.
 3) for k ← 1 to t do
 4)     if F[k] < 1 or F[k] > n
 5)         return false // Невалиден номер на връх от G.
 6) for k ← 1 to t − 1 do
 7)     for ℓ ← k + 1 to t do
 8)         if F[k] = F[ℓ]
 9)             return false // Повторение на върхове.
10)         if k ∈ Adj_T(ℓ) and F[k] ∉ Adj_G(F[ℓ])
11)             return false // Несъответствие на ребра.
12) return true
```

Анализ на алгоритъма за проверка: Най-лошият случай е, когато сертификатът е валиден. Тогава алгоритъмът изпълнява всички проверки на сертификата и приключва чрез ред № 12. Ред № 1 изисква константно време: $\Theta(1)$. Цикълът с начало на ред № 3 изразходва време $\Theta(t)$. Редове № 8 и № 10 се изпълняват общо $\Theta\big(t^2\big)$ пъти заради циклите с начала на редове № 6 и № 7. Всяка отделна проверка на ред № 8 отнема време $\Theta(1)$, но ред № 10 съдържа два скрити цикъла: проверките за принадлежност на елемент към списък изискват обхождане на двата списъка. Единият списък съдържа не повече от m ребра, а другият — не повече от $t-1$ ребра. Затова всяко отделно изпълнение на ред № 10 изразходва време $\mathrm{O}(m+t)$, а общо за целия алгоритъм времето е $\mathrm{O}\big(t^2(m+t)\big)$. Този израз — полином от трета степен — представлява горна граница на времевата сложност на алгоритъма. Следователно Problem2 $\in$ **NP**.

Щом задачата Problem2 е **NP**-трудна и принадлежи на **NP**, то тя е **NP**-пълна.

Задача 3. Задачата "В ориентиран граф G има ли хамилтонов път от върха s до върха t?" остава ли **NP**-трудна, когато графът G е двуделен?

Решение: Да, задачата остава **NP**-трудна. Нека означим с Problem3 случая на двуделен граф, а с "Хамилтонов път" — общия случай (когато ориентираният нетегловен граф G е произволен). Ще сведем общия случай до частния, като преобразуваме дадения произволен граф в двуделен. Формалното описание на редукцията изглежда така:

Хамилтонов път $\big(G$: ориентиран нетегловен граф; s и t: върхове на $G\big)$
1) $\big(\widetilde{G},\, s_{\text{out}}\,,\, t_{\text{in}}\big) \leftarrow$ Превръщане на ориентиран граф в двуделен $\big(G,\, s,\, t\big)$
2) **return** Problem3$\big(\widetilde{G},\, s_{\text{out}}\,,\, t_{\text{in}}\big)$

Превръщането на произволен ориентиран граф G в двуделен $\widetilde{G}$ се извършва по следния начин:
— За всеки от върховете s и t на G създаваме по един нов връх на $\widetilde{G}$ — съответно s_{out} и t_{in}.
— За всеки друг връх v на G създаваме два върха на $\widetilde{G}$ и ги свързваме с ребро: $v_{\text{in}} \to v_{\text{out}}$.
— Пренебрегваме ребрата на G, които влизат в s или излизат от t.
— За всяко друго ребро $u \to v$ на G създаваме ребро $u_{\text{out}} \to v_{\text{in}}$.

Бързина на редукцията: Всеки връх и всяко ребро на G се обработват за константно време. Следователно общото време на редукцията (тоест времето за изпълнение на ред № 1 от кода) е линейно, а оттам и полиномиално спрямо размера на G.

Коректността на редукцията следва от две твърдения:
— Първо, графът $\widetilde{G}$ е двуделен. Това твърдение гарантира, че на задачата Problem3 се подават допустими входни данни. За доказателство е достатъчно да забележим, че върховете на графа $\widetilde{G}$ са оцветени в два цвята — in и out, като краищата на всяко ребро са разноцветни по построение.
— Второ, $\widetilde{G}$ съдържа хамилтонов път от s_{out} до $t_{\text{in}} \iff G$ съдържа хамилтонов път от s до t. В посока отдясно наляво това твърдение се доказва така: ако $sxy\ldots zt$ е хамилтонов път в G, то $s_{\text{out}}x_{\text{in}}x_{\text{out}}y_{\text{in}}y_{\text{out}}\ldots z_{\text{in}}z_{\text{out}}t_{\text{in}}$ е хамилтонов път в $\widetilde{G}$. Обратната импликация се доказва въз основа на това, че всеки хамилтонов път от s_{out} до t_{in} в графа $\widetilde{G}$ редува върхове in и out, като след всеки връх in веднага идва едноименният му връх out (според построението на $\widetilde{G}$). Тоест всеки хамилтонов път от s_{out} до t_{in} в графа $\widetilde{G}$ има вида $s_{\text{out}}x_{\text{in}}x_{\text{out}}y_{\text{in}}y_{\text{out}}\ldots z_{\text{in}}z_{\text{out}}t_{\text{in}}$. Тогава $sxy\ldots zt$ е хамилтонов път в G.

И така, има полиномиална редукция "Хамилтонов път" $\propto_{\text{P}}$ Problem3. От теорията знаем, че "Хамилтонов път" е **NP**-трудна задача. Следователно Problem3 също е **NP**-трудна.

Забележка: Можем да направим графа двуделен, добавяйки връх по средата на всяко ребро. Този начин е приложим и към неориентирани графи, но не върши работа в настоящата задача: $\widetilde{G}$ съдържа хамилтонов път $\iff G$ съдържа път, който е едновременно хамилтонов и ойлеров.