NP-ПЪЛНИ ЗАДАЧИ

ПРИМЕРНО КОНТРОЛНО № 5 ПО "ДИЗАЙН И АНАЛИЗ НА АЛГОРИТМИ" — СУ, ФМИ
(ЗА СПЕЦИАЛНОСТ "КОМПЮТЪРНИ НАУКИ", 1. ПОТОК, ФЕВРУАРИ – ЮНИ 2019 Г.)

Задача 1. Разглеждаме задачата за разпознаване `LongestCycle`:
— Вход: неориентиран тегловен граф G и положително число K.
— Въпрос: G съдържа ли прост цикъл, чиито ребра имат тегла със сбор $\geq K$?
Един цикъл е прост, ако не повтаря върхове.

Докажете, че `LongestCycle` е NP-пълна задача.

Задача 2. Разглеждаме задачата за разпознаване `ManyKnapsacks`:
— Вход: две цели положителни числа K и C;
и масив $A[1...n]$ от цели положителни числа, $n > 0$.
— Въпрос: Може ли $A[1...n]$ да се разбие на K непразни мултимножества,
всяко от които със сбор $\leq C$?

Всеки елемент на масива $A[1...n]$ трябва да принадлежи на едно и само едно мултимножество. Въпреки това говорим за мултимножества, а не за множества, тъй като масивът $A[1...n]$ може да съдържа повторения, тоест елементи, които имат различни индекси, но равни стойности. Поради това може да се случи някое мултимножество да съдържа една и съща стойност няколко пъти; тогава тя участва със същия брой събираеми.

Казано неформално, задачата `ManyKnapsacks` звучи така:
може ли n предмета с дадени тегла $A[1...n]$ да се пренесат с помощта на K раници с еднакви капацитети C, ако във всяка раница трябва да сложим поне един предмет?

Докажете, че `ManyKnapsacks` е NP-пълна задача.

Задача 3. Разглеждаме задачата за разпознаване `NonEmptyRows`:
— Вход: цяло положително число K
и двоична матрица $A[1...n][1...m]$ с m стълба и n реда.
— Въпрос: Съществуват ли K стълба на матрицата A, такива че подматрицата, образувана от тези K стълба и всичките n реда на A, съдържа единица във всеки свой ред?

Докажете, че `NonEmptyRows` е NP-пълна задача.

Задача 4. Разглеждаме задачата за разпознаване `SumMatrix`:
— Вход: цяло число $S \geq 0$ и матрица $B[1...n][1...m]$ с m стълба и n реда, съставена от цели неотрицателни числа.
— Въпрос: Може ли да се изберат n числа, по едно от всеки ред на матрицата B, така, че сборът на избраните n числа да е равен на S ?

Докажете, че `SumMatrix` е NP-пълна задача.

РЕШЕНИЯ

Задача 1. Ще докажем, че `LongestCycle` е NP-трудна задача, с помощта на полиномиална редукция: `HamiltonianCycle` $\propto$ `LongestCycle`, където `HamiltonianCycle` е задачата за разпознаване дали даден неориентиран нетегловен граф G съдържа хамилтонов цикъл, тоест цикъл, който преминава точно веднъж през всеки връх на G.

Описание на редукцията:

```
HamiltonianCycle(G: неориентиран нетегловен граф с n върха)
1) Обхождаме G и даваме тегло 1 на всяко ребро.
2) K ← n
3) return LongestCycle(G, K)
```

Доказателството за коректност на редукцията се състои от две части:

— Параметрите G и K, с които извикваме функцията `LongestCycle`, имат допустими стойности. Действително, $K = n$ е положително число. Графът G отначало е неориентиран и нетегловен, ала след като го обходим и дадем тегло 1 на ребрата му, G става тегловен граф (и остава неориентиран).

— Функцията `HamiltonianCycle` връща правилен резултат. Наистина,

функцията `HamiltonianCycle(G)` връща стойност "истина"

⇕ (от ред № 3 на алгоритъма)

функцията `LongestCycle(G, K)` връща стойност "истина"

⇕ (от определението на задачата `LongestCycle`)

G съдържа прост цикъл, чиито ребра имат тегла със сбор $\geq K$

⇕ (от ред № 2 на алгоритъма: $K = n$)

G съдържа прост цикъл, чиито ребра имат тегла със сбор $\geq n$

⇕ (от ред № 1 на алгоритъма: сборът от теглата на ребрата е техният брой)

G съдържа прост цикъл с поне n ребра

⇕ (за всеки цикъл броят на ребрата е равен на броя на върховете)

G съдържа прост цикъл с поне n върха

⇕ (простият цикъл не повтаря върхове, затова има най-много n върха)

G съдържа прост цикъл с точно n върха

⇕ (от определението на прост цикъл и от n = броя на върховете на G)

G съдържа цикъл, минаващ точно веднъж през всеки връх на G

⇕ (от определението на хамилтонов цикъл)

G съдържа хамилтонов цикъл.

И тъй, функцията `HamiltonianCycle(G)` връща стойност "истина" $\Leftrightarrow$ графът G съдържа хамилтонов цикъл. Това съвпада с определението на задачата `HamiltonianCycle`, следователно редукцията е коректна.

Бързина на редукцията: Редукцията се състои от редове № 1 и № 2 на функцията `HamiltonianCycle`. Ред № 1 се изпълнява за линейно време: $\Theta(m+n)$, където m и n са съответно броят на ребрата и броят на върховете на графа G. Видът на обхождането (в ширина или в дълбочина) няма значение. Ред № 2 се изпълнява за константно време: $\Theta(1)$. Общото време на редукцията е равно на $\Theta(m+n) + \Theta(1) = \Theta(m+n)$. То е линейно, значи полиномиално, следователно редукцията е достатъчно бърза.

С това доказахме, че алгоритмичната задача `LongestCycle` е NP-трудна. Остава да докажем, че тя принадлежи на **NP**, тоест предложено решение може да бъде проверено за полиномиално време. Естествено, решение може да бъде предложено само при отговор "да" на въпроса на задачата, т.е. когато графът G съдържа прост цикъл, чиито ребра имат тегла със сбор ≥ K. Най-естествено е сертификатът да бъде търсеният прост цикъл, напр. списък от върховете му.

Проверката на предложеното решение може да се извърши например така:

```
CheckLongestCycle(
   G: неориентиран нетегловен граф с n върха;
   K: положително число;
   C: списък от върхове с дължина L - предполагаемият цикъл)
1) if L > n
2)    return false
3) if списъкът C съдържа повторения
4)    return false
5) if има два поредни върха от C, между които няма ребро в G
6)    // последният и първият връх от C също са поредни
7)    return false
8) S ← сбора от дължините на ребрата между всеки два
        поредни върха от C, вкл. първия и последния връх
9) return (S ≥ K)
```

С ред № 5 проверяваме дали предложеният цикъл се състои от ребра на графа G, а с ред № 3 проверяваме дали цикълът е прост. С редове № 8 и 9 проверяваме дали сборът S от дължините на ребрата на цикъла C е по-голям или равен на K. Предложеното решение C е правилно тогава и само тогава, когато всички тези проверки минат успешно; в противен случай то е грешно.

Ако $L > n$, то от принципа на Дирихле следва, че цикълът C повтаря върхове на графа G, следователно не може да бъде прост. Това обосновава коректността на редове № 1 и № 2. Алгоритъмът е коректен и без тях, но тяхното наличие спестява време, както ще видим по-долу.

Анализ на бързодействието: От редове № 1 и № 2 следва, че при $L > n$ алгоритъмът `CheckLongestCycle` има константна времева сложност: $\Theta(1)$. При $L \le n$ се изпълняват и другите редове. Проверката на ред № 3 можем да извършим за време $O(L^2)$, например като сравним всеки два елемента на C. Проверката на ред № 5 изисква еднократно обхождане на върховете от C, а за всеки от тях — обхождане на списъка на излизащите от него ребра на G; в най-лошия случай се обхождат всички ребра на G по два пъти (по веднъж за всеки от краищата); това прави общо време $O(L+m)$ за тази проверка. Пресмятането на сбора S от ред № 8 изисква същото време (и може да стане в рамките на същото обхождане, при което се прави проверката от ред № 5). Проверката на неравенството на ред № 9 изразходва константно време: $\Theta(1)$. Общото време на алгоритъма е равно на $O(L^2) + O(L+m) = O(L^2+m)$, което е равно на $O(n^2+m)$, тъй като $L \le n$. (Именно тук са важни редове № 1 и № 2.)

И тъй, `CheckLongestCycle` се изпълнява за не повече от квадратично, следователно полиномиално време. Затова задачата `LongestCycle` е от **NP**.

Щом задачата `LongestCycle` е от **NP** и е NP-трудна, то тя е NP-пълна.

Задача 2. Че `ManyKnapsacks` е NP-трудна задача, следва с помощта на полиномиална редукция: `Partition` $\propto$ `ManyKnapsacks`. Тук `Partition` е задачата за разпознаване дали множество $A[1...n]$ от цели положителни числа може да се разбие на две части с равни сборове.

Описание на редукцията:

```
Partition(A[1...n]: array of positive integers)
1) C ← 0
2) for i ← 1 to n do
3)     C ← C + A[i]
4) if C is odd    // Нечетен сбор не може да се разбие
5)     return false // на два равни сбора от цели числа.
6) C ← C / 2
7) K ← 2
8) return ManyKnapsacks(A[1...n], K, C)
```

Доказателството за коректност на редукцията се състои от две части:

— Функцията `ManyKnapsacks` се извиква само с допустими стойности — цели положителни числа. По-специално, параметърът C е цяло число, защото делението на 2 (на ред № 6) се изпълнява само ако C е било четно преди това.
— Функцията `Partition` връща правилен резултат. Има два случая.

Първи случай: сборът на елементите на входния масив е нечетно число. Тогава променливата C има нечетна стойност на ред № 4 от алгоритъма, затова се изпълнява ред № 5 и алгоритъмът връща false, тоест търсеното разбиване не съществува. Действително, всяка от предполагаемите две части би имала целочислен сбор (тъй като събираемите са цели числа), а тъй като двата сбора трябва да са равни, то C трябва да бъде удвоената стойност на всеки от тях, т.е. четно число. Затова при нечетно C няма разбиване с равни сборове, тоест функцията правилно връща стойност "неистина" (false).

Втори случай: сборът на всички елементи на входния масив е четно число. Тогава разсъждаваме по следния начин:

функцията `Partition(A[1...n])` връща стойност "истина"

⇕ (от ред № 8 на алгоритъма)

`ManyKnapsacks(A[1...n], K, C)` връща стойност "истина"

⇕ (от определението на задачата `ManyKnapsacks`)

`A[1...n]` се разбива на K непразни мултимножества, всяко със сбор ≤ C

⇕ (от ред № 7 на алгоритъма: $K = 2$)

`A[1...n]` се разбива на две непразни мултимножества, всяко със сбор ≤ C

⇕ (от редове № 1 – № 6: C = полусбора на всички числа от масива `A`)

множеството `A[1...n]` се разбива на две непразни мултимножества, всяко със сбор, ненадхвърлящ половината от сбора на всички числа на `A`

⇕ (строго неравенство е невъзможно)

множеството `A[1...n]` се разбива на две непразни мултимножества, всяко със сбор, равен на половината от сбора на всички числа на `A`

⇕

множеството `A[1...n]` се разбива на две непразни мултимножества с равни сборове.

И тъй, функцията `Partition(A[1...n])` връща стойност "истина" ⇔ множеството `A[1...n]` се разбива на две непразни мултимножества с равни сборове. Това съвпада с определението на задачата `Partition`, следователно редукцията е коректна.

Бързина на редукцията: Редукцията се състои от редове № 1 – № 7 на функцията `Partition`. Цикълът от редове № 2 – № 3 пресмята сбора за линейно време: $\Theta(n)$. Другите оператори изискват константно време: $\Theta(1)$. Общото време на редукцията е $\Theta(n)$. То е линейно, значи полиномиално, следователно редукцията е достатъчно бърза.

С това доказахме, че задачата ManyKnapsacks е NP-трудна. Остава да се уверим, че тя принадлежи на **NP**, тоест предложено решение може да бъде проверено за полиномиално време. Решение може да бъде предложено само при отговор "да" на въпроса на задачата, т.е. когато входният масив може да бъде разбит на K мултимножества, всяко със сбор ≤ C. Най-естествено е сертификатът да бъде търсеното разбиване, например масив Parts[1...n], където Parts[i] е номерът на мултимножеството, съдържащо $A[i]$. С други думи, Parts[i] е номерът на раницата, в която е сложен предмет № i, и трябва да е цяло число от 1 до K вкл. Предложено решение може да се провери например така:

```
CheckManyKnapsacks(
   A[1...n], Parts[1...n]: arrays of positive integers;
   K, C: positive integers)
 1) if K > n
 2)    return false
 3) W[1...K]: array of positive integers // total weights
 4) for p ← 1 to K do
 5)    W[p] ← 0
 6) for i ← 1 to n do
 7)    p ← Parts[i]
 8)    if p > K
 9)       return false
10)    W[p] ← W[p] + A[i]
11)    if W[p] > C
12)       return false
13) for p ← 1 to K do
14)    if W[p] = 0
15)       return false
16) return true
```

С ред № 8 проверяваме дали сертификатът съдържа невалидни стойности; ако е така, с ред № 9 обявяваме сертификата за невалиден. В редове № 6 – № 10 натрупваме събираемите в съответния сбор (предмет № i отива в раница № p), след като предварително сме нулирали всички сборове (редове № 4 – № 5). С ред № 11 проверяваме дали теглото на раницата, в която току-що сме сложили предмет, надхвърля капацитета; ако е така, с помощта на ред № 12 отхвърляме предложеното решение. Ако накрая се окаже, че има празна раница, отхвърляме предложеното решение (редове № 13 – № 15). В противен случай предложеното решение изпълнява всички изисквания, затова го приемаме (ред № 16).

Според горния анализ алгоритъмът за проверка на предложено решение е коректен и без редове № 1 и № 2. Той е коректен и с тях, защото при $K > n$ поне една раница е празна, а това не се допуска от условието на задачата. Тоест при $K > n$ е сигурно отнапред, че въпросът на задачата `ManyKnapsacks` има отговор "не", затова е излишно да изследваме предложеното решение.

Редове № 1 и № 2 са сложени с цел бързодействие. Трите последователни цикъла имат времеви сложности $\Theta(K)$ и $\Theta(n)$, поради което времето на целия алгоритъм е $\Theta(K+n)$ в най-лошия случай — когато предложеното решение е вярно. Ако K е много по-голямо от n, времевата сложност $\Theta(K+n) = \Theta(K)$ е псевдолинейна, защото K е стойност, а не дължина на входа. Дължината на числото K (броят на цифрите му) е $\log K$, затова $\Theta(K)$ е експоненциална сложност. Тоест алгоритъмът за проверка на предложено решение би бил бавен при голямо K, ако ги нямаше редове № 1 и № 2. Обаче при наличието на тези два реда изводите от анализа се променят така: при $K > n$ алгоритъмът работи ограничено време $\Theta(1)$, тъй като излиза от ред № 2; а при $K \le n$ сложността му по време е $\Theta(K+n)$, но сега това не е проблем, тъй като $\Theta(K+n) = \Theta(n)$.

Окончателно, алгоритъмът за проверка на предложено решение притежава линейна, следователно полиномиална времева сложност: $\Theta(n)$. Затова задачата `ManyKnapsacks` е от **NP**.

Щом задачата `ManyKnapsacks` е от **NP** и е NP-трудна, то тя е NP-пълна.

Задача 3. Ще докажем, че NonEmptyRows е NP-трудна задача, с помощта на полиномиална редукция: DominatingSet $\propto$ NonEmptyRows, където DominatingSet е задачата за разпознаване дали неориентиран нетегловен граф G има доминиращо множество с не повече от K върха (множество D, такова че всеки връх, който не е от D, е свързан чрез ребро с някой връх от D).

Описание на редукцията:

```
DominatingSet(G: неориентиран нетегловен граф с n върха;
              K: цяло положително число)
1) A[1...n][1...n] ← матрицата на съседствата на G
2) for i ← 1 to n do
3)    A[i][i] ← 1
4) return NonEmptyRows(A[1...n][1...n], K)
```

Доказателството за коректност на редукцията се състои от две части:

— Параметрите A и K, с които извикваме функцията NonEmptyRows, имат допустими стойности: K е цяло число, $K > 0$, а матрицата A е двоична, защото между всеки два върха на граф или има ребро, или няма.

— Функцията DominatingSet връща правилен резултат. Наистина,

функцията DominatingSet(G, K) връща стойност "истина"

⇕ (от ред № 4 на алгоритъма)

функцията NonEmptyRows(A, K) връща стойност "истина"

⇕ (от определението на задачата NonEmptyRows)

съществуват K стълба на матрицата A, такива че подматрицата, образувана от тези K стълба и всичките n реда на A, съдържа единица във всеки свой ред

⇕ (от редове № 2 и № 3 на алгоритъма: $A[i][i] = 1$ за всяко i)

съществуват K стълба на матрицата A, такива че подматрицата, образувана от тези K стълба и другите $n - K$ реда на A, съдържа единица във всеки свой ред ("другите $n - K$ реда" са редовете с различни номера от избраните K стълба)

⇕ (от ред № 1 на алгоритъма)

съществуват K върха на графа G, такива че всеки от другите $n - K$ върха е свързан чрез ребро с някой от тези K върха

⇕ (от определението на доминиращо множество)

G съдържа доминиращо множество с не повече от K върха.

И тъй, функцията DominatingSet(G, K) връща стойност "истина" $\Leftrightarrow$ графът G съдържа доминиращо множество с не повече от K върха. Това съвпада с въпроса на задачата DominatingSet, т.е. редукцията е коректна.

Бързина на редукцията: Редукцията се състои от редове № 1, № 2 и № 3 на функцията `DominatingSet`. Ред № 1 изисква квадратично време: $\Theta\left(n^2\right)$, а цикълът на редове № 2 и № 3 — линейно време $\Theta(n)$. Поради това общото време на редукцията е $\Theta\left(n^2\right)$. То е квадратично, значи полиномиално, тоест редукцията е достатъчно бърза.

Дотук доказахме, че алгоритмичната задача `NonEmptyRows` е NP-трудна. Остава да докажем, че тя принадлежи на **NP**, тоест предложено решение може да бъде проверено за полиномиално време. Решение може да бъде предложено само при отговор "да" — когато A съдържа подходящи K стълба. Сертификат може да бъде множеството от индексите им (списък или логически масив C).

Проверката на предложеното решение може да се извърши например така:

```
CheckNonEmptyRows(A[1...n][1...n], K, C[1...n])
 1) cnt ← 0
 2) for j ← 1 to n do
 3)    if C[j]
 4)       cnt ← cnt + 1
 5) if cnt ≠ K
 6)    return false
 7) for i ← 1 to n do
 8)    has1 ← false
 9)    for j ← 1 to n do
10)       if C[j] and A[i][j] = 1
11)          has1 ← true  // A има единица в ред № i
12)    if not has1
13)       return false    // A няма единица в ред № i
14) return true
```

С редове № 1 – № 6 проверяваме дали предложените стълбове са точно K. С редове № 7 – № 13 проверяваме дали подматрицата, образувана от предложените K стълба, съдържа единица във всеки свой ред. Ако някое от тези две изисквания не е спазено, отхвърляме предложеното решение (редове № 6 и № 13). В противен случай предложеното решение отговаря на всички изисквания, затова го приемаме (ред № 14).

Анализ на бързодействието: Поради двата вложени цикъла проверката на сертификата изисква време, което е най-много квадратично: $\mathrm{O}\left(n^2\right)$, следователно полиномиално. Затова задачата `NonEmptyRows` е от **NP**.

Задача 4. Ще докажем, че SumMatrix е NP-трудна задача, с помощта на полиномиална редукция: SubsetSum $\propto$ SumMatrix, където SubsetSum е задачата за разпознаване дали дадено цяло неотрицателно число S може да се образува като сбор от някои елементи на даден масив $A[1...n]$ от цели положителни числа.

Описание на редукцията:

```
SubsetSum(A[1...n], S)
1) B[1...n][1...2]: array of non-negative integers // m = 2
2) for k ← 1 to n do
3)     B[k][1] ← A[k]
4)     B[k][2] ← 0
5) return SumMatrix(B[1...n][1...2], S)
```

Доказателството за коректност на редукцията се състои от две части:

— Параметрите B и S, с които извикваме функцията SumMatrix, имат допустими стойности: S е цяло число, $S \geq 0$, а матрицата B се състои от цели неотрицателни числа.

— Функцията SubsetSum връща правилен резултат. Наистина,
функцията SubsetSum(A, S) връща стойност "истина"
⇕ (от ред № 5 на алгоритъма)
функцията SumMatrix(B, S) връща стойност "истина"
⇕ (от определението на задачата SumMatrix)
числото S може да се образува като сбор от n числа,
взети по едно от всеки ред на матрицата B
⇕ (от ред № 4 на алгоритъма: нулевите събираеми не влияят на сбора)
числото S може да се образува като сбор от няколко числа,
взети от първия стълб на матрицата B
⇕ (от ред № 3 на алгоритъма)
числото S може да се образува като сбор от няколко елемента на масива A.

И тъй, функцията SubsetSum(A[1...n], S) връща стойност "истина" $\Leftrightarrow$ числото S е сбор от няколко елемента на масива A. Това твърдение съвпада с определението на задачата SubsetSum, тоест редукцията е коректна.

Бързина на редукцията: Редукцията се състои от редовете № 1 – № 4 на функцията SubsetSum. Цикълът изисква време $\Theta(n)$, което е линейно, следователно полиномиално. Това значи, че редукцията е достатъчно бърза за целите на доказателството.

Дотук доказахме, че алгоритмичната задача `SumMatrix` е NP-трудна. Остава да докажем, че тя принадлежи на **NP**, тоест предложено решение може да бъде проверено за полиномиално време. Решение може да бъде предложено само при отговор "да", т.е. когато матрицата *B* съдържа подходящи *n* числа. Най-естествено е сертификатът да бъде масив от индекси на стълбове *C* [1...*n*]: *C* [*i*] е номерът на стълба, от който е взето събираемото в *i*-тия ред на *B*.

Проверката на предложеното решение може да се извърши например така:

```
CheckSumMatrix(B[1...n][1...m]: non-negative integers
               S: non-negative integer;
               C[1...n]: array of positive integers)
1) sum ← 0
2) for i ← 1 to n do
3)    j ← C[i]
4)    if j > m
5)       return false
6)    sum ← sum + B[i][j]
7) return (sum = S)
```

С ред № 4 проверяваме дали сертификатът съдържа невалиден индекс на стълб. С цикъла събираме предложените елементи на матрицата *B*. Накрая (ред № 7) проверяваме дали се получава желаният сбор *S*.

Анализ на бързодействието: Поради цикъла проверката на сертификата изисква време, което е най-много линейно: $O(n)$, следователно полиномиално. Затова задачата `SumMatrix` е от **NP**.

Щом задачата `SumMatrix` е от **NP** и е NP-трудна, то тя е NP-пълна.

СХЕМА ЗА ТОЧКУВАНЕ

Всяка задача носи 25 точки, разпределени по следния начин:
— за описание на редукцията: 5 точки;
— за доказателство на коректността на редукцията: 5 точки;
— за анализ на бързодействието на редукцията: 5 точки;
— за съставяне на алгоритъм с полиномиална сложност за проверка на предложено решение (сертификат): 5 точки;
— за анализ на бързодействието на алгоритъма за проверка: 5 точки.

Тоест 15 точки са за доказване, че алгоритмичната задача е NP-трудна, а 10 точки се дават за доказателство, че тя принадлежи на класа NP.