ИЗПИТ ПО УЧЕБНАТА ДИСЦИПЛИНА "ДИЗАЙН И АНАЛИЗ НА АЛГОРИТМИ"
(СОФИЙСКИ УНИВЕРСИТЕТ "СВЕТИ КЛИМЕНТ ОХРИДСКИ", ФМИ, 07. 02. 2020 Г.)

Указания:
1) Решенията на задачите да бъдат описани и обосновани подробно.
2) Названията на алгоритмите и структурите от данни да бъдат на български език.
3) Когато задача се решава чрез граф, да се опише графът — ориентиран или неориентиран е; какво са върховете и ребрата му; какво значат посоките и теглата на ребрата (ако има такива).
4) При обхождане на граф да се уточнява видът на обхождането — в ширина или в дълбочина. При извършване на сортиране или търсене да се уточнява името на използвания алгоритъм.
5) Ако някоя задача се решава чрез графи, алгоритъмът да се опише словесно. Да не се съставя алгоритъм "от нулата". Вместо това да се използват наготово алгоритмите, изучени в курса.
6) Ако някоя задача се решава с динамично програмиране, алгоритъмът да се опише на псевдокод.
7) За редукциите е задължително да има псевдокод, обосновка за коректност и анализ на времето.
8) Да се анализира времевата сложност на всеки предложен алгоритъм.
9) Във всички задачи изискваната времева сложност се отнася за най-лошия случай.

Задача 1. Имаме редица от n величини $\alpha_1, \alpha_2, \ldots, \alpha_n$, например температура, налягане и т.н. Да предположим, че всеки две от тези величини са силно корелирани. Корелациите са два вида:
— положителна корелация: когато едната величина расте, другата също расте (и обратно);
— отрицателна корелация: когато едната величина расте, другата намалява.

Дадена е матрица с n реда и n стълба, чиито елементи са два вида — плюсове и минуси. В пресечната точка на ред № i и стълб № j стои знакът на корелацията между величините α_i и α_j.

Съществува ли алгоритъм с линейна времева сложност, който проверява има ли противоречие в матрицата? Ако да — съставете такъв алгоритъм и анализирайте неговата времева сложност. Ако не — докажете подходяща долна граница.

Приемете без проверка, че матрицата е симетрична и по главния диагонал има само плюсове.

Задача 2. Даден е числов масив $A[1...n]$, чиито елементи са едноцифрени числа от 1 до 9 вкл. (без нулата). Дадено е и едно цяло число F, което може да е положително, отрицателно или нула. Съставете алгоритъм, който за време $O(n^2)$ разпознава дали е възможно да бъдат поставени знаци плюс и минус пред числата така, че да се получи алгебричен сбор със стойност, равна на F.

Ако алгоритъмът има сложност по памет $O(n)$ и времева сложност $O(n^2)$ в най-лошия случай, решението се оценява с допълнителни 10 точки.

Още 10 точки се дават за алгоритъм, който при утвърдителен отговор отпечатва самите знаци и има времева сложност $O(n^2)$ в най-лошия случай без ограничения за сложността по памет.

Задача 3. Разглеждаме алгоритмичната задача за разпознаване `SubsetZero`:
Вход: масив $A[1...n]$ от n цели числа с произволни знаци.
Въпрос: Съществува ли непразен сбор 0 от някои от дадените числа (вкл. едно или всички)?

По-формално: Съществува ли непразно множество $M \subseteq \{1, 2, \ldots, n\}$, такова че $\sum_{k \in M} A[k] = 0$?

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

Задача 4. Програма за дешифриране се опитва да разгадае даден текст $A[1...n]$ — низ от n знака, всеки от които има една от 256 възможни стойности (тоест текстът е зададен във формат ASCII). Програмата сравнява дадения низ $A[1...n]$ с други низове, взети от някакъв списък (напр. речник). Естествено, програмистът иска да ускори колкото може повече всяка отделна проверка. Той знае, че текстът $A[1...n]$ е бил шифрован (от някой от текстовете в списъка) само с помощта на една пермутация и една субституция.

Пермутация наричаме всяко разместване на буквите в текста. Например от думата "стол" можем с помощта на пермутация да получим думата "лост". Идентитетът (липсата на разместване) също е пермутация, тоест всяка дума е пермутация на себе си.

Субституция наричаме всяка недвусмислена размяна на значенията на символите в ASCII. Например буквата "а" ще се чете като "б", "б" — като "в", . . . , "ю" — като "я", "я" — като "а". При тази конкретна субституция думата "лют" ще се чете като "мяу", "цяр" — като "час", "цял" — като "чам" и тъй нататък. Всяка субституция е пак пермутация, но на знаците в ASCII, а не на знаците в думата. Идентитетът (липсата на размяна на значенията) също е субституция, тоест всяка дума е субституция на себе си.

Помогнете на програмиста, като съставите алгоритъм, който за време $O(n)$ в най-лошия случай проверява дали два дадени низа $A[1...n]$ и $B[1...n]$ могат да се получат един от друг с помощта на пермутация и субституция. Низовете имат еднаква дължина n и се състоят от знаци в код ASCII, затова можете да приемете, че $A[1...n]$ и $B[1...n]$ са масиви от n цели числа между 0 и 255 вкл.

Задача 5. На черната дъска е написано числото 12345678987654321. Двама играчи се редуват. Всеки, който е на ход, може да намали произволно избрана цифра с една или две единици, при условие че редицата от цифри остава правилен запис на цяло положително число в десетична бройна система (не се допускат отрицателни цифри, а най-лявата цифра не може да бъде нула). Който не може да направи ход, губи играта. Кой от двамата има печеливша стратегия и каква е тя?

Точките се удвояват за строго доказателство, че играта завършва и че описаната стратегия е печеливша. Формалното доказателство, че стратегията е печеливша, трябва да съдържа два инварианта — свойство, което числото на дъската има след всеки ход на първия играч, и друго свойство, което числото има след всеки ход на втория играч. Инвариантите да се докажат. Формалното доказателство, че играта завършва, трябва да е основано на подходящ полуинвариант. От полуинварианта и двата инварианта да се изведат строго твърденията, че играта завършва и че съответният играч печели.

СХЕМА НА ОЦЕНЯВАНЕ

Всяка пълно решена задача носи 20 точки. Оценката от задачи се получава по следната схема:
от 0 до 39 точки — слаб (2);
от 40 до 54 точки — среден (3);
от 55 до 69 точки — добър (4);
от 70 до 84 точки — много добър (5);
от 85 точки нагоре — отличен (6).

РЕШЕНИЯ

Задача 1. Строим пълен неориентиран нетегловен граф с n върха — величините $\alpha_1, \alpha_2, \ldots, \alpha_n$. Ребрата съответстват на корелациите. Всяко ребро има знак — плюс или минус, според вида на корелацията. Подграфът G, съставен от отрицателните ребра, трябва да бъде двуделен: ако в матрицата няма противоречие, то трябва да можем да оцветим върховете на G в два цвята — посоките на изменение на величините (цвят "расте" и цвят "намалява"). Проверката за двуделност извършваме с едно обхождане на G (няма значение дали в ширина, или в дълбочина).

Ако G не е двуделен граф, то матрицата съдържа противоречие и алгоритъмът приключва. В противен случай алгоритъмът преминава към следващата стъпка.

Положителната корелираност е релация на еквивалентност, затова подграфът H, образуван от положителните ребра на пълния граф, трябва да бъде обединение на клики. Затова търсим компонентите на слаба свързаност на H чрез едно обхождане на този подграф (няма значение дали в ширина, или в дълбочина). Проверяваме всяка компонента съдържа ли отрицателни ребра. Ако намерим отрицателно ребро в някоя компонента на слаба свързаност на H, правим извод, че матрицата съдържа противоречие: върховете от една компонента съответстват на величини, които са положително корелирани, затова между тях не може да има отрицателни ребра.

Ако и тази проверка мине успешно, то матрицата не съдържа противоречие.

Построяването на графа и обхождането на неговите подграфи изразходва време $\Theta(n+m)$, където $m = \Theta(n^2)$ е броят на ребрата на пълния граф с n върха. Следователно $n + m = \Theta(n^2)$, т.е. времевата сложност на предложения алгоритъм е линейна, защото $\Theta(n^2)$ е дължината на входа (броят на елементите на матрицата).

Задача 1 може да се реши и без явно позоваване на графи. Матрицата не съдържа противоречие тогава и само тогава, когато можем да припишем посоки на изменение на всички величини ("расте" или "намалява") по съгласуван начин. Понеже всеки две величини са силно корелирани, то приписването на посоки на изменение може да стане (ако изобщо може) само по два начина: единият от тях е негатив на другия. Затова не е ограничение да приемем, че величината α_1 расте. Да припишем стойност $+1$ на растящите величини и стойност -1 на намаляващите. Да приемем, че дадената матрица $M = (m_{ij})$ се състои не просто от знаци плюс и минус, а от числа $+1$ и -1 съответно. От известните правила за умножение следва, че за да показва правилно корелациите, матрицата M трябва да представлява своеобразна таблица на умножението, тоест трябва

$$m_{ij} = \alpha_i \alpha_j \text{ за всяка двойка индекси } i \text{ и } j \text{ (вкл. при } i = j).$$

Тъй като приехме, че $\alpha_1 = +1$, то $m_{i1} = \alpha_i$ и $m_{1j} = \alpha_j$ за всички допустими индекси. Затова матрицата не съдържа противоречие тогава и само тогава, когато

$$m_{ij} = m_{i1}\, m_{1j} \text{ за всяка двойка индекси } i \text{ и } j.$$

Проверката на това равенство става с помощта на два вложени цикъла. Излишно е да проверяваме случаите, когато някой от индексите има стойност 1, тъй като всички тези проверки се свеждат до равенството $m_{11} = +1$, което е изпълнено по условие (елементите върху главния диагонал са положителни).

```
CheckMatrix(M[1...n][1...n]: array of +1 and -1): Boolean
1) for i ← 2 to n do
2)    for j ← 2 to n do
3)       if M[i][j] ≠ M[i][1] × M[1][j]
4)          return false
5) return true
```

Заради двата вложени цикъла този алгоритъм също има линейна времева сложност: $\Theta(n^2)$. Тоест двата алгоритъма са еднакво бързи по порядък. Обаче новият алгоритъм използва паметта по-икономично: за двата брояча е нужна памет само $\Theta(1)$, а за отбелязване на посетените върхове при обхождане на граф е нужна памет $\Theta(n)$. Затова алгоритъмът `CheckMatrix` носи още 20 т. (тоест той се оценява с 40 точки).

Отговорът "не съществува алгоритъм с линейна времева сложност" носи нула точки.

Задача 2 може да се реши за полиномиално време с помощта на ***динамично програмиране***. За целта решаваме същата задача за всички по-къси масиви и за всички суми от $-9n$ до $+9n$ вкл. Не можем да се ограничим само със суми между 0 и F, тъй като може да се наложи да излезем извън този интервал. От друга страна, тъй като събираемите са едноцифрени и са n на брой, то стойността на всеки възможен алгебричен сбор се намира между $-9n$ и $+9n$ включително. Можем да спестим малко памет, като разглеждаме само сумите между $-S$ и $+S$ включително, където S е сборът от всички елементи на дадения масив; така паметта намалява в повечето случаи, но не и в най-лошия случай — когато входният масив се състои само от деветки.

Псевдокод:

```
CanPutSigns1(A[1...n]: array of decimal digits; F: integer): Boolean
 1) S ← 0
 2) for k ← 1 to n do
 3)    S ← S + A[k]
 4) if |F| > S
 5)    return false
 6) dyn[0...n][-S...S]: array of Boolean
 7) // dyn[ ñ ][ F̃ ] = истина ⇔ числото F̃ може да бъде стойност
 8) // на алгебричен сбор, образуван от първите ñ елемента на A.
 9) for F̃ ← -S to S do
10)    dyn[0][ F̃ ] ← false
11) dyn[0][0] ← true  // Празната сума е равна на нула.
12) for ñ ← 1 to n do
13)    for F̃ ← -S to S do  // Непълно булево изчисление.
14)       if F̃ + A[ ñ ] ≤ S and dyn[ ñ - 1][ F̃ + A[ ñ ]]
15)          dyn[ ñ ][ F̃ ] ← true
16)       else if F̃ - A[ ñ ] ≥ -S and dyn[ ñ - 1][ F̃ - A[ ñ ]]
17)          dyn[ ñ ][ F̃ ] ← true
18)       else
19)          dyn[ ñ ][ F̃ ] ← false
20) return dyn[n][F]
```

Анализ на сложността: Сложността по памет е $\Theta(nS)$ заради динамичната таблица `dyn`. Сложността по време е също $\Theta(nS)$ заради вложените цикли, започващи от редове № 12 и № 13. Понеже $S = 9n$ в най-лошия случай (когато входният масив A се състои само от деветки), то в крайна сметка сложността и по време, и по памет е $\Theta(n^2)$. Тя ще остане по порядък толкова дори без малката оптимизация със сбора S. Съответният псевдокод се получава от кода по-горе, като изтрием първите три реда, а в останалите редове заменим S с $9n$.

Тъй като `dyn[ ñ ][-S...S]` зависи само от `dyn[ ñ - 1][-S...S]`, то достатъчно е да пазим само два последователни реда от динамичната таблица. Така ще намалим порядъка на количеството памет, използвана от алгоритъма. Получаваме втора версия на алгоритъма.

```
CanPutSigns2(A[1...n]: array of decimal digits; F: integer): Boolean
 1) S ← 0
 2) for k ← 1 to n do
 3)    S ← S + A[k]
 4) if |F| > S
 5)    return false
 6) dyn[0...1][-S...S]: array of Boolean
 7) // dyn[1][ F̃ ]: текущият ред ( ñ ) от таблицата на CanPutSigns1;
 8) // dyn[0][ F̃ ]: предишният ред ( ñ - 1 ) от споменатата таблица.
 9) for F̃ ← -S to S do
10)    dyn[1][ F̃ ] ← false
11) dyn[1][0] ← true  // Празната сума е равна на нула.
12) for ñ ← 1 to n do
13)    for F̃ ← -S to S do // Копираме новия ред на мястото на стария.
14)       dyn[0][ F̃ ] ← dyn[1][ F̃ ]
15)    for F̃ ← -S to S do  // Непълно булево изчисление.
16)       if F̃ + A[ ñ ] ≤ S and dyn[0][ F̃ + A[ ñ ]]
17)          dyn[1][ F̃ ] ← true
18)       else if F̃ - A[ ñ ] ≥ -S and dyn[0][ F̃ - A[ ñ ]]
19)          dyn[1][ F̃ ] ← true
20)       else
21)          dyn[1][ F̃ ] ← false
22) return dyn[1][F]
```

Анализ на сложността: Сложността по памет е $\Theta(S)$ заради динамичната таблица `dyn`. Сложността по време е $\Theta(nS)$ заради вложените цикли с начала на редове № 12, № 13 и № 15. Понеже $S = 9n$ в най-лошия случай (когато входният масив A се състои само от деветки), то в крайна сметка сложността по памет е $\Theta(n)$, а по време е $\Theta(n^2)$.

Коя да е от тези две версии може да бъде допълнена така, че да намира и самите събираеми, образуващи алгебричен сбор със стойност F. За тази цел трябва всеки път, когато записваме стойност "истина" в динамичната таблица, да пазим още и знака на последното събираемо в сбора. Следователно ще бъде нужна една допълнителна таблица с размерите на динамичната таблица. В новата таблица ще записваме знаците на събираемите. Всеки знак е или плюс, или минус, затова новата таблица може да пази стойности от логически тип: истина — когато знакът е плюс; лъжа — когато знакът е минус.

Възстановяването на знаците на числата става в ред, обратен на попълването на таблиците: най-напред възстановяваме знака на последното събираемо с помощта на допълнителната таблица; после от текущата сума и последното нейно събираемо (със знак) намираме чрез изваждане сбора на останалите събираеми и взимаме този сбор като нова текуща сума. Така определяме знаците един по един, докато изчерпим всички събираеми.

```
PutSigns(A[1...n]: array of decimal digits; F: integer): Boolean
 1) S ← 0
 2) for k ← 1 to n do
 3)    S ← S + A[k]
 4) if |F| > S
 5)    return false
 6) dyn[0...n][-S...S]: array of Boolean
 7) sgn[1...n][-S...S]: array of Boolean
 8) // dyn[ ñ ][ F̃ ] = истина ⇔ числото F̃ може да бъде стойност
 9) // на алгебричен сбор, образуван от първите ñ елемента на A.
10) // sgn[ ñ ][ F̃ ] = истина ⇔ последното събираемо има знак плюс.
11) for F̃ ← -S to S do
12)    dyn[0][ F̃ ] ← false
13) dyn[0][0] ← true  // Празната сума е равна на нула.
14) for ñ ← 1 to n do
15)    for F̃ ← -S to S do  // Непълно булево изчисление.
16)       if F̃ + A[ ñ ] ≤ S and dyn[ ñ - 1][ F̃ + A[ ñ ]]
17)          dyn[ ñ ][ F̃ ] ← true
18)          sgn[ ñ ][ F̃ ] ← false
19)       else if F̃ - A[ ñ ] ≥ -S and dyn[ ñ - 1][ F̃ - A[ ñ ]]
20)          dyn[ ñ ][ F̃ ] ← true
21)          sgn[ ñ ][ F̃ ] ← true
22)       else
23)          dyn[ ñ ][ F̃ ] ← false
24) if dyn[n][F]  // Възстановяване на знаците в обратен ред.
25)    F̃ ← F
26)    for ñ ← n downto 1 do
27)       if sgn[ ñ ][ F̃ ]
28)          print "плюс"
29)          F̃ ← F̃ - A[ ñ ]
30)       else
31)          print "минус"
32)          F̃ ← F̃ + A[ ñ ]
33) return dyn[n][F]
```

Анализ на сложността: Сложността по памет е $\Theta(nS) = \Theta(n^2)$ заради таблиците `dyn` и `sgn`. И сложността по време е $\Theta(nS) = \Theta(n^2)$ заради вложените цикли с начала на редове № 14 и № 15. Можем да спестим памет от таблицата `dyn`, но не и от таблицата `sgn`. Тоест можем да намалим константния множител пред порядъка на паметта, но не и самия порядък.

Задача 3. Че `SubsetZero` е NP-трудна задача, се доказва с помощта на полиномиална редукция от известната NP-пълна задача `SubsetSum`:

```
SubsetSum(B[1...n], S)
1) A[1...n+1]: array of integers;
2) for k ← 1 to n do
3)   A[k] ← B[k] // A[k] > 0
4) A[n+1] ← -S   // A[n+1] ≤ 0
5) return SubsetZero(A[1...n+1])
```

Коректност на редукцията: В приведения код $B[1...n]$ е масив от n цели положителни числа, а пък S е цяло неотрицателно число.

Функцията `SubsetSum(B[1...n], S)` връща истина.

$\Updownarrow$ (от ред № 5)

`SubsetZero(A[1...n+1])` връща истина.

$\Updownarrow$ (от определението на задачата `SubsetZero`)

Съществува непразно множество $M \subseteq \{1, 2, \ldots, n+1\}$, такова че $\sum_{k \in M} A[k] = 0$.

$\Updownarrow$ (от знаците на числата `B[k]` и `S` следва, че $n+1 \in M$)

$\exists M \subseteq \{1, 2, \ldots, n+1\}$: $n+1 \in M$ и $\sum_{k \in M} A[k] = 0$.

$\Updownarrow$ ($M = \widetilde{M} \cup \{n+1\}$)

$\exists \widetilde{M} \subseteq \{1, 2, \ldots, n\}$: $A[n+1] + \sum_{k \in \widetilde{M}} A[k] = 0$.

$\Updownarrow$ (от редове № 3 и № 4)

$\exists \widetilde{M} \subseteq \{1, 2, \ldots, n\}$: $-S + \sum_{k \in \widetilde{M}} B[k] = 0$.

$\Updownarrow$ (прехвърляме S)

$\exists \widetilde{M} \subseteq \{1, 2, \ldots, n\}$: $\sum_{k \in \widetilde{M}} B[k] = S$.

Доказахме, че функцията `SubsetSum(B[1...n], S)` връща истина точно когато числото S може да се представи като сбор на някои (вкл. всички или нито едно) от числата `B[1...n]`. Това съвпада с определението на задачата `SubsetSum`, следователно редукцията е коректна.

Бързина на редукцията: Редукцията се състои от редове № 2, № 3 и № 4, които се изпълняват за общо време $\Theta(n)$, следователно редукцията е полиномиална.

И така, доказахме, че съществува полиномиална редукция `SubsetSum` $\propto$ `SubsetZero`. Знаем, че задачата `SubsetSum` е NP-трудна. Значи, задачата `SubsetZero` също е NP-трудна.

Че задачата `SubsetZero` принадлежи на класа NP, се доказва, като съставим алгоритъм с полиномиална времева сложност за проверка на предложено решение. Сертификат ще бъде множеството M, представено като логически масив.

```
CheckSubsetZero(A[1...n], M[1...n])
sum ← 0
cnt ← 0
for k ← 1 to n do
  if M[k]
    cnt ← cnt + 1
    sum ← sum + A[k]
return (cnt > 0) and (sum = 0)
```

Този алгоритъм има полиномиална времева сложност, защото се изпълнява за време $\Theta(n)$.

Щом задачата `SubsetZero` е NP-трудна и принадлежи на класа NP, то тя е NP-пълна.

Задача 4. Използваме идеята на ***сортирането чрез броене.*** За да унищожим пермутациите, броим колко пъти се среща всеки знак в единия и в другия низ. Така получаваме два масива от абсолютните честоти на знаците. Ако нямаше субституции, двата масива щяха да бъдат равни. Субституциите разместват елементите на масивите с бройките. За да унищожим субституциите, е достатъчно да сортираме масивите с бройките. Ако след това двата масива станат равни, то дадените низове могат да се получат един от друг чрез пермутация и субституция (и обратно).

Псевдокод:

```
CheckCypher(A[1...n], B[1...n]: arrays of ASCII characters): Boolean
C[0...255], D[0...255]: arrays of integers
for k ← 0 to 255 do
  C[k] ← 0
  D[k] ← 0
for k ← 1 to n do
  C[A[k]] ← C[A[k]] + 1
  D[B[k]] ← D[B[k]] + 1
Sort(C[0...255])
Sort(D[0...255])
for k ← 0 to 255 do
  if C[k] ≠ D[k]
    return false  // Двата низа A и B не се получават един от друг.
return true   // Двата низа A и B могат да се получат един от друг
// с помощта на пермутация и субституция.
```

Анализ на времевата сложност: Първият и третият цикъл изразходват време $\Theta(1)$, а вторият цикъл — време $\Theta(n)$. Независимо какви сортировки използваме за масивите C и D, те отнемат време $\Theta(1)$, защото дължината на всеки от тези масиви е константа (256), тоест не зависи от n. Окончателно, времето на целия алгоритъм е $\Theta(n)$.

Задача 5 е дадена на Задочната олимпиада по комбинаторика, проведена през април-май 2019 г. от Московския физико-технически институт. Адрес на страницата на олимпиадата в Интернет: http://polyanskii.com/other/combolymp/?fbclid=IwAR1_XRoy_jWDTqpoOp--w3-OFOEKR5BmXbgkvUmEM38y0njYj5ojch40_3I
Архив на условията на задачите има и на адрес: https://learn.fmi.uni-sofia.bg/mod/resource/view.php?id=122280

При правилна игра печели първият играч:
— На първия ход променя цифрата на единиците от 1 на 0.
— На всеки следващ ход играе симетрично на последния ход на втория играч:
  — ако вторият играч е намалил средната цифра с x, то първият играч я намалява с $3 - x$;
  — ако вторият играч е намалил някоя друга цифра с x, то първият играч намалява с x цифрата, която ѝ е симетрична относно средата на числовия низ. **(20 точки)**

Полуинвариант е числото, записано на черната дъска, защото то се изменя строго монотонно: намалява след всеки ход. Тъй като то винаги е цяло положително, а не съществува безкрайна строго намаляваща редица от цели положителни числа, то играта ще завърши, тоест някой играч ще остане без ход. Играта завършва с числото 10000000000000000: само то не може да се намали по правилата на играта. **(4 точки)**

През цялото време на играта най-лявата цифра е 1: тя е 1 в началото и няма как да се намали (числото не може да започва с цифрата 0). Аналогично, най-дясната цифра става 0 след първия ход на първия играч, след което не може да бъде намалявана повече. Тези две цифри не участват повече в играта, затова не се интересуваме от тях. Останалите цифри разделяме в кошници: във всяка кошница слагаме по две цифри, симетрични относно средната цифра, а тя отива сама в отделна кошница. Тази кошница наричаме правилна, ако средната цифра се дели на 3. За всяка от другите кошници казваме, че е правилна, ако двете цифри в нея са равни.

За всички цифри без първата и последната важат следните инварианти:
— След всеки ход на първия играч всички кошници са правилни. **(2 точки)**
— След всеки ход на втория играч има точно една неправилна кошница и ако тя се състои от две цифри, то разликата им е 1 или 2. **(2 точки)**

Доказателство на инвариантите: чрез взаимна индукция.

База: След първия ход на първия играч на дъската е написано числото 12345678987654320. След като махнем първата и последната цифра (които не ни интересуват), остава числовият низ 234567898765432. Средната му цифра 9 се дели на 3, а всеки две цифри, които са симетрични относно средната цифра, са равни: $2 = 2$, $3 = 3$, . . . , $8 = 8$. **(2 точки)**

Индуктивна стъпка: Нека първият инвариант важи след някой непоследен ход на първия играч. Ще докажем, че след първия следващ ход на втория играч ще бъде изпълнен вторият инвариант. За хода на втория играч има две възможности — да намали средната или някоя друга цифра. Ако вторият играч намали някоя друга цифра, нейната симетрична цифра ще остане непроменена. Според индуктивното предположение двете цифри са били равни преди този ход. Затова след него те ще се различават с 1 или 2 и тази кошница ще стане неправилна. Ако вторият играч намали средната цифра с 1 или 2, то пак от индуктивното предположение следва, че след намаляването тя ще дава остатък съответно 2 или 1 при деление на 3, т.е. нейната кошница ще стане неправилна. Другите кошници остават непроменени, значи правилни според индуктивното предположение. Затова след хода на втория играч има точно една неправилна кошница. **(4 точки)**

Нека вторият инвариант важи след някой непоследен ход на втория играч. Ще докажем, че след първия следващ ход на първия играч ще бъде изпълнен първият инвариант. От индуктивното предположение знаем, че преди хода на първия играч има точно една неправилна кошница. Стратегията му поправя тази кошница, затова след неговия ход всички кошници ще са правилни. Поправянето е възможно, защото неправилната кошница е само една и ако съдържа двойка цифри, разликата им е 1 или 2, а ако съдържа средната цифра, остатъкът е 1 или 2 по модул 3. **(4 точки)**

Доказахме, че играта завършва с числото 10000000000000000. Махаме цифрите в двата края, остават само нули и всички кошници са правилни. Вторият инвариант не важи, значи това число не е получено след ход на втория играч. Следователно числото 10000000000000000 се получава, тоест играта завършва, след ход на първия играч и той печели. **(2 точки)**