Изпит по "Дизайн и анализ на алгоритми" (редовна сесия — СУ, ФМИ, юни 2016 г.)

Име: ______________________________ ФН:_____ Спец.:____ Курс:___

Задача	1а	1б	2а	2б	3	4а	4б	5	Общо
получени точки									
максимум точки	20	20	10	10	20	40	10	20	150

Забележка: За отлична оценка са достатъчни 100 точки.

Задача 1. Дадени са цяло положително число k и неориентиран свързан тегловен граф G с положителни тегла на ребрата.

а) Съставете бърз алгоритъм за построяване на минимална покриваща гора на G с k дървета.

б) Докажете коректността на алгоритъма.

Задача 2. Експертна система разполага с m доказателства на n теореми. Всяко доказателство представлява извеждане на една теорема от друга. Системата умее да комбинира известните ѝ доказателства: ако от теоремата X следва теоремата Y, а от теоремата Y следва теоремата Z, то от теоремата X следва теоремата Z.

За всеки от следните проблеми предложете алгоритъм с линейна времева сложност $\Theta(m+n)$:

а) По две дадени теореми системата трябва да определи дали едната теорема следва от другата; ако да, тогава трябва да се състави доказателство чрез комбиниране на минимален брой от известните m доказателства.

б) Множеството от n теореми да се разбие на подмножества, всяко от които се състои от еквивалентни теореми.

Задача 3. Докажете, че алгоритмичната задача за търсене на стойност *key* в сортиран масив $A[1 \dots n]$ чрез алгоритъм, основан на сравнения, има времева сложност $\Omega(\log n)$.

Упътване: Използвайте дърво за взимане на решения или игра срещу противник.

Задача 4. Математиците в една държава въвели собствена парична единица — непер. От тази валута има само банкноти с номинал 1, 7 и 10 непера.

а) Съставете алгоритъм, който намира най-малкия брой банкноти, с които може да се изплати сумата n непера точно (без ресто).

За скорост на алгоритъма $O(n)$ ще получите 20 точки, а за скорост $O(1)$ — 40 точки.

б) Разширете алгоритъма така, че да предлага начин на плащане с минимален брой банкноти.

Забележка: Демонстрирайте работата на всички предложени алгоритми при $n = 15$ непера.

Задача 5. Да се докаже, че е NP-трудна следната алгоритмична задача:
"Дадени са целите положителни числа k, a_1, a_2, $\dots$, a_n и цялото число $S \geq 0$. Да се провери съществува ли множество $M \subseteq \{1, 2, 3, \dots, n\}$, такова, че $\sum_{i \in M} (a_i)^k = S$."

РЕШЕНИЯ

Задача 1.

а) Минимална покриваща гора с k дървета може да се построи с помощта на следната модификация на алгоритъма на Крускал: прекратяваме алгоритъма предсрочно — след присъединяване на $(n-k)$-тото ребро. В този миг построената от алгоритъма гора съдържа точно k дървета. Може да се докаже, че тя е минимална.

б) Една покриваща гора на графа G ще наричаме хубава, ако множеството от нейните ребра е подмножество на множеството от ребрата на някоя минимална покриваща гора на G с k дървета. Тъй като добавянето на ребро към покриваща гора може само да намали броя на дърветата, то всяка хубава покриваща гора има поне k дървета.

Нека $F_p\Big(V, E_p\Big)$ е хубава покриваща гора на G с p дървета, $p > k$. По определение има минимална покриваща гора $\overset{*}{F}\Big(V, \overset{*}{E}\Big)$ с k дървета, такава, че $E_p \subseteq \overset{*}{E}$. Понеже $p > k$, то $E_p \subset \overset{*}{E}$, т.е. включването е строго. Следователно $\overset{*}{E} \setminus E_p \neq \varnothing$. Нека A е множеството от онези ребра на G, всяко от които, добавено към E_p, не затваря цикъл. Такива има: щом $\overset{*}{F}$ е гора, то тя не съдържа цикъл, следователно $A \supseteq \overset{*}{E} \setminus E_p \neq \varnothing$, т.е. $A \neq \varnothing$. Множеството A е крайно (разглеждаме само крайни графи), следователно съдържа ребро $\{u, v\}$ с минимално тегло. Добавяме $\{u, v\}$ към текущото множество от ребра: $E_{p-1} = E_p \cup \Big\{\{u, v\}\Big\}$. Графът $F_{p-1}\Big(V, E_{p-1}\Big)$ е хубава покриваща гора с $p-1$ дървета.

Доказателство: Добавянето на реброто $\{u, v\}$ не затваря цикъл, затова $F_{p-1}\Big(V, E_{p-1}\Big)$ не съдържа цикли, т.е. това е гора. Тя е покриваща, защото съдържа всички върхове на G. Понеже реброто $\{u, v\}$ не затваря цикъл, то върховете u и v са от различни дървета на F_p и добавянето на $\{u, v\}$ слива двете дървета, т.е. броят на дърветата намалява с единица. Следователно F_{p-1} е гора с $p-1$ дървета. Остава да докажем, че гората F_{p-1} е хубава. Ако $\{u, v\} \in \overset{*}{E}$, то $E_{p-1} \subseteq \overset{*}{E}$ и от определението на хубава гора следва, че F_{p-1} е такава. Нека $\{u, v\} \notin \overset{*}{E}$. Разглеждаме два случая.

Първи случай: върховете u и v са от различни дървета на $\overset{*}{F}$. Следователно добавянето на реброто $\{u, v\}$ към $\overset{*}{E}$ води до сливане на две от дърветата на $\overset{*}{F}$, т.е. получава се покриваща гора с $k-1$ дървета. Понеже $\overset{*}{E} \setminus E_p \neq \varnothing$, то има ребро $\{x, y\} \in \overset{*}{E} \setminus E_p$. Премахването на реброто $\{x, y\}$ ще увеличи с единица броя на дърветата, т.е. ще се получи друга покриваща гора с k дървета: $\overset{\circ}{F}\Big(V, \overset{\circ}{E}\Big)$, където $\overset{\circ}{E} = \Big(\overset{*}{E} \cup \Big\{\{u, v\}\Big\}\Big) \setminus \Big\{\{x, y\}\Big\}$.

Втори случай: върховете u и v са от едно и също дърво на $\overset{*}{F}$. Тогава $\overset{*}{E} \cup \{u, v\}$ има цикъл. Не може всички други ребра на цикъла да са от E_p, защото добавянето на $\{u, v\}$ към E_p не затваря цикъл в F_p. Следователно цикълът съдържа ребро $\{x, y\} \in \overset{*}{E} \setminus E_p$. Премахването на реброто $\{x, y\}$ премахва цикъла, т.е. получава се друга покриваща гора с k дървета: $\overset{\circ}{F}\Big(V, \overset{\circ}{E}\Big)$, където $\overset{\circ}{E} = \Big(\overset{*}{E} \cup \Big\{\{u, v\}\Big\}\Big) \setminus \Big\{\{x, y\}\Big\}$.

По-нататък разсъжденията са общи за двата случая.

Както $\overset{*}{F}\left(V,\overset{*}{E}\right)$, така и $\overset{\circ}{F}\left(V,\overset{\circ}{E}\right)$ са покриващи гори с k дървета, но $\overset{*}{F}$ е минимална гора от този вид, затова теглото ѝ удовлетворява неравенството $w\left(\overset{*}{E}\right) \leq w\left(\overset{\circ}{E}\right)$. От формулата $\overset{\circ}{E} = \left(\overset{*}{E} \cup \Big\{\{u,v\}\Big\}\right) \setminus \Big\{\{x,y\}\Big\}$ следва, че $w\left(\overset{\circ}{E}\right) = w\left(\overset{*}{E}\right) + w\Big(\{u,v\}\Big) - w\Big(\{x,y\}\Big)$. След заместване неравенството приема вида $w\Big(\{u,v\}\Big) \geq w\Big(\{x,y\}\Big)$.

От $\{x,y\} \in \overset{*}{E} \setminus E_p$ и $\overset{*}{E} \setminus E_p \subseteq A$ следва, че реброто $\{x,y\} \in A$. Обаче реброто $\{u,v\}$ беше избрано като ребро от A с минимално тегло. Следователно $w\Big(\{u,v\}\Big) \leq w\Big(\{x,y\}\Big)$.

От последните две неравенства следва, че $w\Big(\{u,v\}\Big) = w\Big(\{x,y\}\Big)$. Оттук се получава равенството $w\left(\overset{*}{E}\right) = w\left(\overset{\circ}{E}\right)$. Щом $\overset{*}{F}$ и $\overset{\circ}{F}$ са покриващи гори с k дървета и $\overset{*}{F}$ е минимална такава гора, то от току-що доказаното равенство следва, че $\overset{\circ}{F}$ също е минимална.

Щом $E_p \subset \overset{*}{E}$ и $E_{p-1} = E_p \cup \Big\{\{u,v\}\Big\}$, то
$\overset{\circ}{E} = \left(\overset{*}{E} \cup \Big\{\{u,v\}\Big\}\right) \setminus \Big\{\{x,y\}\Big\} \supseteq \left(E_p \cup \Big\{\{u,v\}\Big\}\right) \setminus \Big\{\{x,y\}\Big\} = E_{p-1} \setminus \Big\{\{x,y\}\Big\}$, т.е. $\overset{\circ}{E} \supseteq E_{p-1} \setminus \Big\{\{x,y\}\Big\}$.

Понеже реброто $\{x,y\} \in \overset{*}{E} \setminus E_p$, то $\{x,y\} \in \overset{*}{E}$, но $\{x,y\} \notin E_p$. Тъй като $\{u,v\} \notin \overset{*}{E}$, то $\{x,y\} \neq \{u,v\}$. Следователно $\{x,y\} \notin E_p \cup \Big\{\{u,v\}\Big\}$, т.е. $\{x,y\} \notin E_{p-1}$, откъдето следва, че $E_{p-1} \setminus \Big\{\{x,y\}\Big\} = E_{p-1}$. Ето защо $\overset{\circ}{E} \supseteq E_{p-1}$.

И така, $E_{p-1} \subseteq \overset{\circ}{E}$ и $\overset{\circ}{F}\left(V,\overset{\circ}{E}\right)$ е минимална покриваща гора с k дървета. По определение $F_{p-1}\Big(V,E_{p-1}\Big)$ е хубава гора, което трябваше да се докаже.

С помощта на доказаното твърдение вече не е трудно да установим, че модифицираният алгоритъм на Крускал работи коректно. Действително, алгоритъмът построява крайна редица от покриващи гори: $F_n\Big(V,E_n\Big)$, $F_{n-1}\Big(V,E_{n-1}\Big)$, $\ldots$, $F_k\Big(V,E_k\Big)$. Графът е свързан (и краен), значи притежава минимална покриваща гора с k дървета. Множеството от нейните ребра е надмножество на $E_n = \varnothing$, защото празното множество е подмножество на всяко множество. Следователно първата покриваща гора F_n е хубава. Според твърдението, доказано преди малко, ако F_p е хубава гора, то и F_{p-1} е такава. Следва, че всички гори от редицата са хубави.

Спираме вниманието си на последната гора от редицата: $F_k\Big(V,E_k\Big)$. Тя е хубава, значи съществува минимална покриваща гора $\overset{*}{F}\left(V,\overset{*}{E}\right)$ с k дървета, такава, че $E_k \subseteq \overset{*}{E}$.

Да допуснем, че е налице строго включване: $E_k \subset \overset{*}{E}$. Тогава можем, както по-горе, да намерим ребро $\{u,v\}$, което, добавено към гората F_k, поражда хубава гора F_{k-1} с $k-1$ дървета. Но това е невъзможно: както казахме, всяка хубава гора има поне k дървета.

Следователно направеното допускане не е вярно. Остава само една възможност: $E_k = \overset{*}{E}$. Тогава F_k съвпада с $\overset{*}{F}$. Понеже $\overset{*}{F}$ е минимална покриваща гора с k дървета, то и F_k е такава. Алгоритъмът връща F_k, т.е. връща минимална покриваща гора с k дървета, следователно е коректен.

Задача 2. Данните могат да бъдат представени чрез насочен граф с n върха и m ребра: върховете съответстват на теоремите, а ребрата — на доказателствата.

а) Извод с най-малък брой стъпки на една теорема от друга съответства на най-къс път от един даден връх до друг даден връх (където дължината на пътя е равна на броя на ребрата). Такъв път може да бъде намерен чрез търсене в ширина.

б) Еквивалентните теореми следват една от друга. Съответстват им силно свързани върхове (всеки от които може да бъде достигнат от другия). На класовете от еквивалентни теореми съответстват компонентите на силна свързаност, които могат да бъдат намерени чрез известния алгоритъм, извършващ обхождане в дълбочина.

И двата предложени алгоритъма имат линейна времева сложност $\Theta(m+n)$.

Задача 3 може да се реши по различни начини. Както и да бъде решавана, съществено е разсъжденията да се отнасят за <u>всички</u> възможни алгоритми, а не за един конкретен алгоритъм!

Първи начин: чрез дърво за взимане на решения.

Всяко сравнение може да се разглежда като въпрос с два възможни отговора — "да" или "не". Например сравнението "Вярно ли е, че $key < A[k]$?" допуска два възможни отговора, а именно: "Да, $key < A[k]$." или "Не, $key \geq A[k]$." В програмен код на такъв въпрос съответства разклонение **if — then — else**.

На всеки алгоритъм (при всяко n) съответства дърво за взимане на решения, което описва изпълнението на алгоритъма (при конкретната стойност на n). Във всеки връх (без листата) стои въпрос (сравнение) и излизат две ребра, съответстващи на двата отговора — "да" и "не". Тоест имаме двоично кореново дърво. В листата стоят възможните позиции на key, тоест числата от 1 до n, плюс специалното съобщение "key липсва в масива A". На всеки връх можем да съпоставим и едно подмножество на $\left\{1, 2, 3, \ldots, n\right\}$, а именно — множеството от индексите на онези елементи на масива A, които биха могли да съдържат стойността key към текущия момент от изпълнението на алгоритъма. В корена на двоичното дърво стои пълното множество $\left\{1, 2, 3, \ldots, n\right\}$.

Тъй като всяко сравнение изисква константно време, то за цялото изпълнение на алгоритъма е нужно време, равно по порядък на броя на сравненията, т.е. на броя на ребрата в пътя от корена до листото, съответстващо на входните данни. Времевата сложност $T(n)$ е времето за изпълнение в най-лошия случай, а то е равно по порядък на дължината h на най-дълъг път от корена до листата:

$$T(n) \asymp h.$$

Числото h се нарича височина на дървото. Двоично дърво с височина h има най-много 2^h листа. Дървото в нашата задача има поне $n+1$ листа — целите числа от 1 до n и специалната стойност "key липсва в масива A". Ето защо

$$2^h \geq n+1, \quad \text{тоест} \quad h \geq \log_2(n+1).$$

Оттук следва редица от равенства и неравенства:

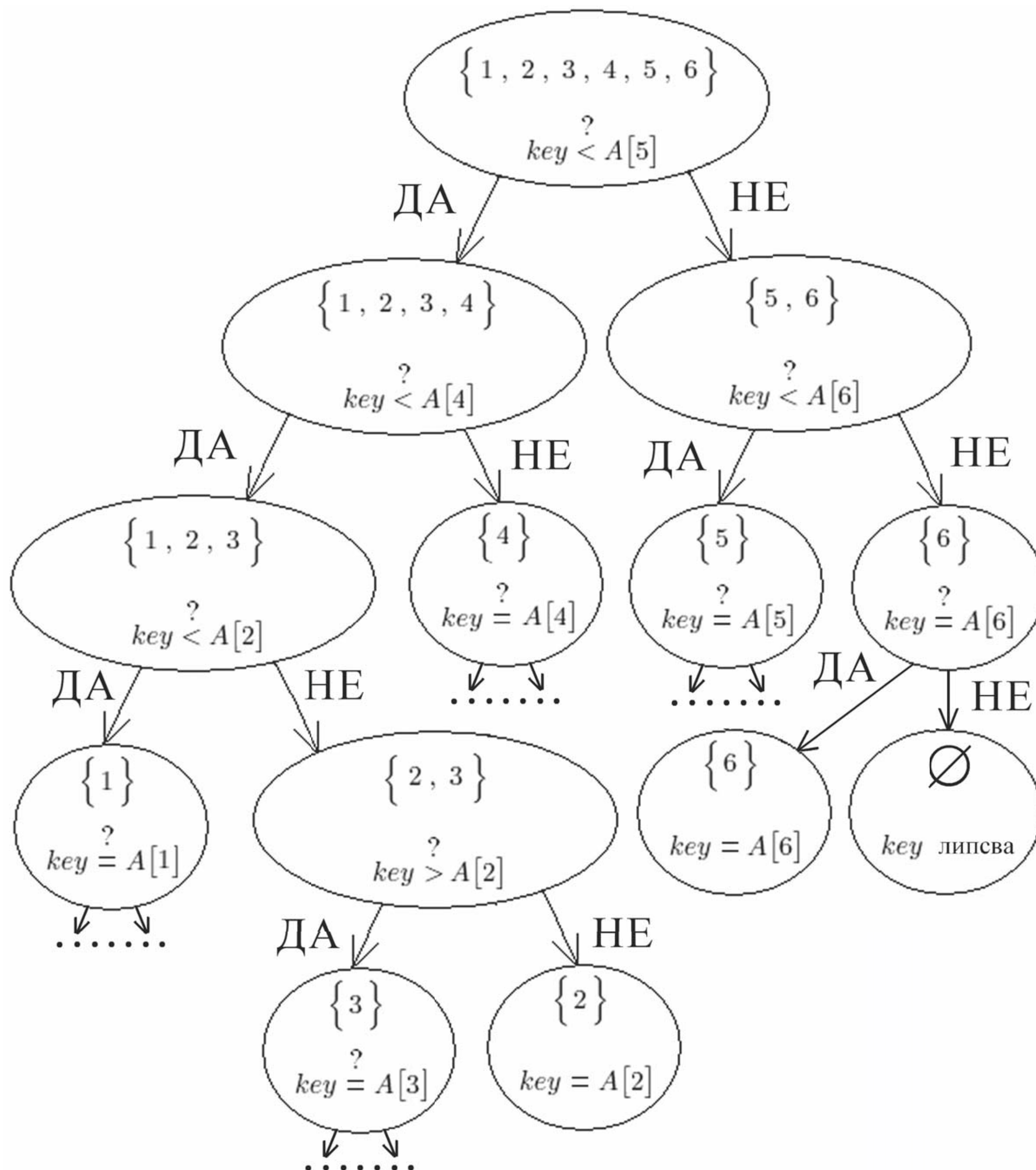
$$T(n) \asymp h \geq \log_2(n+1) \asymp \log n.$$

Като изоставим междинните членове, получаваме асимптотичното неравенство

$$T(n) \succeq \log n,$$

което трябваше да се докаже.

П р и м е р : Да разгледаме изпълнението на един конкретен алгоритъм при $n = 6$. За да подчертаем, че в конкретния алгоритъм няма нищо специално, съзнателно избираме алгоритъм, различен от стандартното двоично търсене.



Многоточията заместват разклонения на дървото, които не са изобразени, защото са подобни на други, вече показани клонове.

Някои листа имат множество от вида $\{k\}$ и равенство $key = A[k]$, например $key = A[2]$. Това е не сравнение, а констатация: търсената стойност key е намерена на k-тата позиция в масива A. Други листа съдържат празното множество и констатацията "key липсва".

Забележка: Възможно е малко по-различно тълкуване на сравненията: можем да смятаме, че всяко сравнение е въпрос от вида "Какъв е знакът между числата x и y ?", а отговорът е един от трите знака "по-голямо", "по-малко" и "равно". Тогава се получава троично дърво и навсякъде в решението основата 2 на степените и на логаритмите се заменя с 3.

Втори начин: чрез игра срещу противник.

По време на играта едно непразно множество $M \subseteq \left\{ 1\ ,\ 2\ ,\ 3\ ,\ \ldots\ ,\ n \right\}$ постоянно се променя. Това M представлява множеството от индексите на онези елементи на масива A, които биха могли да съдържат стойността key към текущия момент. Ето защо играта започва с множеството $M = \left\{ 1\ ,\ 2\ ,\ 3\ ,\ \ldots\ ,\ n \right\}$. Алгоритъмът и неговият противник се редуват, като първият ход е на алгоритъма.

Всяко сравнение разбива множеството M на две подмножества X и Y; едното съдържа индексите на елементите, удовлетворяващи сравнението, а другото — индексите на елементите, които не го удовлетворяват. "Разбиване" означава, че $X \cap Y = \varnothing$ (закон за непротиворечието), и $X \cup Y = M$ (закон за изключеното трето). Позволяваме участието на празни подмножества в разбиванията (такива разбивания съответстват на сравнения с отнапред известен резултат, т.е. излишни сравнения; позволяваме ги за общност на разсъжденията — за да обхванем всички възможни алгоритми, включително и някои, които са очевидно неоптимални).

Изборът на едно или друго сравнение зависи от алгоритъма, т.е. той избира начина на разбиване на текущото множество M.

Резултатът от сравнението зависи от входните данни, т.е. от противника. С други думи, противникът избира X или Y да бъде новото множество M.

Алгоритъмът приключва работа, когато намери търсената стойност (или установи, че тя липсва в масива). Тогава множеството M остава с един или нула елемента.

Правила на играта:
1) Играта започва с множеството $M = \left\{ 1\ ,\ 2\ ,\ 3\ ,\ \ldots\ ,\ n \right\}$.
2) Двамата играчи — алгоритъмът и неговият противник — се редуват.
3) Алгоритъмът играе първи.
4) На всеки свой ход алгоритъмът избира две множества X и Y,
такива, че $X \cap Y = \varnothing$ и $X \cup Y = M$.
5) На всеки свой ход противникът избира или X, или Y да бъде новото M.
6) Играта приключва, когато множеството M остане с един елемент или без елементи.

Целта на алгоритъма е играта да свърши възможно най-бързо.
Целта на противника е да удължи играта колкото може повече.

Забележки по правилата:

— В правило № 6 се допуска играта да завърши и тогава, когато множеството M остане с един елемент. Тогава са възможни два случая: алгоритъмът може да е сигурен, че този елемент е търсеният индекс (т.е. намерил е търсената стойност в масива), но може и да не е сигурен (защото е възможно масивът изобщо да не съдържа търсената стойност). Във втория случай се налага едно допълнително сравнение, а такова не се предвижда от правилата на играта. Това не е проблем, защото всяко сравнение изисква константно време, което може да бъде пренебрегнато, щом правим оценка по порядък.

— В правило № 4 се допуска произволно разбиване, което е много повече, отколкото може да се постигне чрез едно сравнение. (Например няма как чрез едно сравнение да се разделят числата от 1 до 10 на четни и нечетни.) Това, че даваме на алгоритъма повече възможности, не е проблем, защото така задачата се решава за по-широк клас от алгоритми.

Тъй като всяко сравнение изразходва константно време, то времето за изпълнение на целия алгоритъм е правопропорционално, следователно равно по порядък на броя на сравненията, т.е. на броя на ходовете в играта. Понеже се интересуваме от времевата сложност в най-лошия случай, то достатъчно е да покажем, че за всяко n съществуват входни данни, които изискват поне $\Omega(\log n)$ сравнения. Това е равносилно на съществуването на стратегия на противника, която му позволява да продължи играта поне $\Omega(\log n)$ хода.

Такава стратегия се построява лесно: тъй като по време на играта множеството M намалява, а целта на противника е играта да продължи максимално дълго, то той трябва да се постарае да забави намаляването на множеството M. За целта противникът трябва на всеки свой ход да избира по-голямото от множествата X и Y (ако са еднакво големи, няма значение кое от тях ще избере).

Тази стратегия гарантира, че на всеки ход множеството M намалява най-много два пъти. Следователно за два хода M намалява най-много четири пъти, за три хода — осем пъти и т.н. По индукция следва, че за k хода множеството M намалява най-много 2^k пъти. Като следва описаната стратегия, противникът никога не избира празно множество, така че играта завършва, когато M остане с един елемент, т.е. намалее n пъти. Ако това стане за k хода, то $n \leq 2^k$, следователно $k \geq \log_2 n$, което трябваше да се докаже.

Задача 4 може да се реши по различни начини.

Първи начин: чрез динамично програмиране.

а) Ако се интересуваме само от минималния общ брой банкноти, е достатъчно да го пазим в един масив `total` и да увеличаваме бройката с единица при добавянето на всяка банкнота.

```
ALG_1A (n)
1  total[−9 ... n]: array of integers   // минимален общ брой банкноти
2  for k ← −9 to −1
3       total[k] ← +∞
4  total[0] ← 0
5  for k ← 1 to n
6       total[k] ← 1 + min ( total[k − 1] , total[k − 7] , total[k − 10] )
7  return total[n]
```

Демонстрация на алгоритъма при $n = 15$ непера:

k	-9	...	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`total`	$+\infty$	...	$+\infty$	0	1	2	3	4	5	6	1	2	3	1	2	3	4	2	3

Извод: За сумата $n = 15$ непера са нужни най-малко три банкноти.

Сложността на алгоритъма по време и памет е $\Theta(n)$. Количеството допълнителна памет (но не и времето на работа) може да бъде намалено по порядък благодарение на това, че новата стойност в масива зависи най-много от десет предишни. Следователно можем да пазим само тях, т.е. можем да решим задачата с константен брой променливи от примитивен тип.

б) Алгоритъмът може да бъде разширен така, че да връща не само общия брой банкноти, но и броя на банкнотите от всеки вид. За краткост нека функцията `min` връща наредена двойка, чийто първи елемент е най-малката от стойностите, а втори елемент е поредният ѝ номер. С други думи, `min(p,q,r)` връща `(p,1)` или `(q,2)`, или `(r,3)`.

```
Alg_1в (n)
 1  total[−9...n]: array of integers   // минимален общ брой банкноти
 2  ones[0...n]: array of integers   // оптимален брой банкноти от 1 непер
 3  sevens[0...n]: array of integers   // оптимален брой банкноти от 7 непера
 4  for k ← −9 to −1
 5      total[k] ← +∞
 6  total[0] ← 0
 7  ones[0] ← 0
 8  sevens[0] ← 0
 9  for k ← 1 to n
10      ( total[k] , kind ) ← min ( total[k − 1] , total[k − 7] , total[k − 10] )
11      total[k] ← total[k] + 1
12      switch
13          case kind = 1 :   ones[k] ← ones[k − 1] + 1 ;   sevens[k] ← sevens[k − 1]
14          case kind = 2 :   ones[k] ← ones[k − 7] ;       sevens[k] ← sevens[k − 7] + 1
15          case kind = 3 :   ones[k] ← ones[k − 10] ;      sevens[k] ← sevens[k − 10]
16  print "Брой банкноти от 10 непера: ", total[n] − ones[n] − sevens[n]
17  print "Брой банкноти от 7 непера: ", sevens[n]
18  print "Брой банкноти от 1 непер: ", ones[n]
19  return total[n]
```

За коректността на алгоритъма няма значение коя най-малка стойност връща функцията `min` при наличие на няколко такива. За определеност приемаме, че в този случай функцията `min` връща последната по ред най-малка стойност, т.е. алгоритъмът предпочита банкноти с по-голям номинал.

Демонстрация на алгоритъма при $n = 15$ непера:

k	-9	...	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`total`	$+\infty$	...	$+\infty$	0	1	2	3	4	5	6	1	2	3	1	2	3	4	2	3
`ones`				0	1	2	3	4	5	6	0	1	2	0	1	2	3	0	1
`sevens`				0	0	0	0	0	0	0	1	1	1	0	0	0	0	2	2

Извод: За сумата $n = 15$ непера са нужни най-малко три банкноти:
две банкноти от 7 непера и една банкнота от 1 непер.

Сложността на алгоритъма по време и памет е $\Theta(n)$. Количеството допълнителна памет (но не и времето на работа) може да бъде намалено по порядък благодарение на това, че новата стойност във всеки масив зависи най-много от десет предишни. Следователно можем да пазим само тях; така задачата се решава с константен брой променливи от примитивен тип.

Втори начин: Има алгоритъм със сложност $\Theta(1)$ по време и памет. Достатъчни са следните съображения: десет банкноти от 7 непера могат да се заменят със седем банкноти от 10 непера, а седем банкноти от 1 непер могат да бъдат заменени с една банкнота от 7 непера. Следователно всяко решение с най-малък брой банкноти съдържа не повече от девет банкноти от 7 непера и не повече от шест банкноти от 1 непер. Получават се общо $10 \times 7 = 70$ варианта, които могат да бъдат проверени за време, независещо от n.

```
Alg_2 (n)
 1  minTotal ← +∞
 2  for cnt1 ← 0 to 6
 3      for cnt7 ← 0 to 9
 4          r ← n − cnt1 − 7 × cnt7
 5          if r ≥ 0 and r mod 10 = 0
 6              cnt10 ← r / 10
 7              cntTotal ← cnt1 + cnt7 + cnt10
 8              if cntTotal < minTotal
 9                  opt1 ← cnt1
10                  opt7 ← cnt7
11                  opt10 ← cnt10
12                  minTotal ← cntTotal
13  print "Брой банкноти от 10 непера: ", opt10
14  print "Брой банкноти от 7 непера: ", opt7
15  print "Брой банкноти от 1 непер: ", opt1
16  return minTotal
```

При всяко n алгоритъмът изпробва всичките 70 варианта. При $n = 15$ алгоритъмът намира два варианта, когато r е неотрицателно и кратно на 10:
1) две банкноти от 7 непера и една от 1 непер (общо три банкноти);
2) нула банкноти от 7 непера, пет от 1 непер и една от 10 непера (общо шест банкноти).
Първият вариант е по-добър, т.е. нужни са минимум три банкноти.

Трети начин: Въпреки че Alg_2 е най-бърз по порядък (тъй като има константна сложност), той не е най-бързият възможен алгоритъм: константата може да бъде намалена още. Не е нужно при всяко n да бъдат изпробвани всичките 70 случая. Щом оптималното решение съдържа най-много шест банкноти от 1 непер и девет банкноти от 7 непера, то общата стойност на тези банкноти е не повече от $9 \times 7 + 6 \times 1 = 69$ непера. Тогава банкнотите от 10 непера имат обща стойност поне $n - 69$ непера, т.е. поне $n - 69$, закръглено нагоре до най-близкото число, кратно на 10. Остатъкът (не повече от 69 непера) се изплаща по таблица, съдържаща оптималните начини за образуване на сумите от 0 до 69 непера. Тази таблица не се изчислява всеки път. Тя е пресметната еднократно (без значение как) и е записана в кода на алгоритъма.

Пример: При $n = 15$ непера оптималното решение се взима наготово от таблицата: нужни са минимум три банкноти (две от 7 непера и една от 1 непер).

Пример: При $n = 247$ непера: $n - 69 = 178$. Закръглено нагоре до кратно на 10, дава 180 непера, т.е. 18 банкноти от 10 непера. Остатъкът $247 - 180 = 67$ непера се изплаща по таблицата: шест банкноти от 10 непера и една от 7 непера. Получава се минимален брой двайсет и пет банкноти — двайсет и четири от 10 непера и една от 7 непера.

```
Alg_3 (n)
 1  cnt10 ← max(0 ; ⌈(n − 69) / 10⌉)
 2  r ← n − 10 × cnt10      // 0 ≤ r ≤ 69
 3  // начало на таблицата
 4  switch
 5      case r = 0 :  cnt1 ← 0;   cnt7 ← 0   // cnt10 не се променя
 6      ......................................................
 7      case r = 15:  cnt1 ← 1;   cnt7 ← 2   // cnt10 не се променя
 8      ......................................................
 9      case r = 67:  cnt1 ← 0;   cnt7 ← 1;   cnt10 ← 6 + cnt10
10      ......................................................
11      case r = 69:  cnt1 ← 2;   cnt7 ← 1;   cnt10 ← 6 + cnt10
12  // край на таблицата
13  cntTotal ← cnt10 + cnt7 + cnt1
14  print "Брой банкноти от 10 непера: ", cnt10
15  print "Брой банкноти от  7 непера: ", cnt7
16  print "Брой банкноти от  1 непер: ", cnt1
17  return cntTotal
```

Този алгоритъм е по-бърз от Alg_2 (макар и не по порядък), защото при всяко n разглежда само един случай, а не всичките седемдесет. Недостатък на Alg_3 е дългият програмен код.

Четвъртият и петият начин са оптимизации съответно на втория и третия алгоритъм. С малко по-внимателни разсъждения можем да намалим броя на случаите доста под седемдесет. Например няма смисъл да използваме повече от две банкноти от 7 непера, защото три банкноти от 7 непера могат да се заменят с две банкноти от 10 непера и една банкнота от 1 непер: $3 \times 7 = 21 = 2 \times 10 + 1 \times 1$. Така едно оптимално решение (с три банкноти) се заменя с друго оптимално решение (със същия брой банкноти). Ограничавайки до 0, 1 или 2 броя на банкнотите от седем непера, губим някои от оптималните решения, но не всички. Това е допустимо, понеже търсим едно оптимално решение, а не всички такива.

Можем да ограничим и броя на банкнотите от един непер. Броят им е не повече от шест (защото седем банкноти от един непер могат да се заменят с една банкнота от седем непера), и то само ако няма банкноти от седем непера. А ако има, то броят на банкнотите от един непер е максимум две (защото три банкноти от един непер и една от седем непера могат да се заменят с една банкнота от десет непера).

Накратко: Ако cnt1 и cnt7 са съответно броят на банкнотите от един и седем непера, то за всяко n съществува оптимално решение измежду следните наредени двойки (cnt7; cnt1): (0; 0), (0; 1), (0; 2), (0; 3), (0; 4), (0; 5), (0; 6); (1; 0), (1; 1), (1; 2); (2; 0), (2; 1), (2; 2).

Алгоритъмът Alg_4 е вариант на Alg_2, променен така, че да изпробва само тези тринайсет случая вместо всичките седемдесет.

Алгоритъмът Alg_5 е оптимизация на Alg_3 с таблица от 0 до 16 вместо от 0 до 69. Границата 16 се определя от наредената двойка (2; 2): $2 \times 7 + 2 \times 1 = 16$.

Шести начин: чрез поправен алчен алгоритъм.

В желанието си да съставим бързодействащ и къс програмен код е естествено да изпробваме алчен алгоритъм:

1) Използваме колкото може повече банкноти от 10 непера.

2) Като остане сума, по-малка от 10 непера, използваме (ако може) една банкнота от 7 непера.

3) Остатъка (ако има такъв) изплащаме с банкноти от 1 непер.

За съжаление, този алгоритъм е грешен. Например при $n = 15$ той намира представяне с шест банкноти (10+1+1+1+1+1 = 15) вместо с минималния брой три (7+7+1 = 15).

За щастие, алгоритъмът може да бъде поправен. За да се ориентираме, нека разгледаме базовите случаи. Според доказаното по-горе е достатъчно да разгледаме сумите от 0 до 16 непера. За удобство при следващите разсъждения ще разгледаме сумите от 0 до 20 непера.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Минимален общ брой банкноти	0	1	2	3	4	5	6	1	2	3	1	2	3	4	2	3	4	2	3	4	2
Брой банкноти от 10 непера	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	2
Брой банкноти от 7 непера	0	0	0	0	0	0	0	1	1	1	0	0	0	0	2	2	2	1	1	1	0
Брой банкноти от 1 непер	0	1	2	3	4	5	6	0	1	2	0	1	2	3	0	1	2	0	1	2	0

Не е трудно да се види, че от случаите, изброени в таблицата, алчният алгоритъм греши само при $n = 14$, 15 и 16. Понеже използва банкноти от 10 непера, докогато е възможно, той греши при всички стойности на n, завършващи на 4, 5 или 6 (и различни от 4, 5 и 6). В останалите случаи алчният алгоритъм работи коректно.

Следователно алчният алгоритъм може да бъде поправен така: ако числото n завършва на някоя от цифрите 4, 5 и 6 (и е различно от 4, 5 и 6), то трябва да използваме една банкнота от 10 непера по-малко, а вместо нея си служим с две банкноти от 7 непера. Остатъка (ако има) изплащаме с банкноти от 1 непер.

```
ALG_6 (n)
 1  cnt10 ← ⌊n/10⌋   // използваме колкото може повече банкноти от 10 непера
 2  if cnt10 > 0 and n завършва на цифрата 4, 5 или 6
 3      cnt10 ← cnt10 − 1   // поправка
 4  r ← n − 10 × cnt10   // оставаща сума
 5  cnt7 ← ⌊r/7⌋   // използваме колкото може повече банкноти от 7 непера
 6  cnt1 ← r − 7 × cnt7   // остатъка изплащаме с банкноти от 1 непер
 7  print "Брой банкноти от 10 непера: ", cnt10
 8  print "Брой банкноти от  7  непера: ", cnt7
 9  print "Брой банкноти от  1  непер: ", cnt1
10  return cnt10 + cnt7 + cnt1
```

При $n = 15$ отначало cnt10 = 1, но след поправката cnt10 става 0, така че сумата 15 непера се изплаща само с банкноти от 1 и 7 непера: $2 \times 7 + 1 \times 1$ — общо три банкноти, което е минимумът.

Когато n завършва на 0, 1, 2 или 3, няма поправка: използваме колкото може повече банкноти от 10 непера. В тези случаи остатъкът е по-малък от 7 непера, затова се изплаща с банкноти от 1 непер.

Примери:
$253 = 250 + 3 = 25 \times 10 + 3 \times 1$ — общо 28 банкноти.
$461 = 460 + 1 = 46 \times 10 + 1 \times 1$ — общо 47 банкноти.

Когато n завършва на 7, 8 или 9, пак няма поправка: използваме колкото може повече банкноти от 10 непера. Сега остатъкът е поне 7 непера, затова освен банкнотите от 1 непер има и една от 7 непера.

Примери:
$519 = 510 + 9 = 51 \times 10 + 1 \times 7 + 2 \times 1$ — общо 54 банкноти.
$367 = 360 + 7 = 36 \times 10 + 1 \times 7$ — общо 37 банкноти.

Когато n завършва на 4, 5 или 6, има поправка: банкнотите от 10 непера са с една по-малко. Вместо нея използваме две банкноти от 7 непера. Евентуалния остатък изплащаме с банкноти от 1 непер.

Примери:
$106 = 90 + 16 = 9 \times 10 + 2 \times 7 + 2 \times 1$ — общо 13 банкноти.
$875 = 860 + 15 = 86 \times 10 + 2 \times 7 + 1 \times 1$ — общо 89 банкноти.

Алгоритъмът Alg_6 бе предложен от хон. ас. Стефан Фотев. Този алгоритъм е оптимален: има константна сложност, константата е минимална (алгоритъмът не разглежда излишни случаи) и програмният код е възможно най-къс.

Задача 5. В частния случай $k = 1$ се получава известната NP-трудна задача SubsetSum. Редукцията е полиномиална, защото присвояването $k = 1$ се извършва за константно време. Тоест разглежданата алгоритмична задача е обобщение на NP-трудната задача SubsetSum и значи също е NP-трудна.