ДИЗАЙН И АНАЛИЗ НА АЛГОРИТМИ — ВТОРА ЧАСТ
(ИЗПИТ — 10 ФЕВРУАРИ 2021 Г., СУ, ФМИ)

Задача 1. Докажете, че алгоритмичната задача "Дали дадена променлива на дадена компютърна програма при даден вход ще приеме дадена стойност когато и да било по време на изпълнение на програмата?" е нерешима.

Задача 2. Търсим редакционното разстояние (разстоянието на Левенщейн) между думите ДАГЕСТАН и АРЕСТАНТ, тоест най-малкия брой операции (изтривания, вмъквания и замени на знаци), чрез които единият низ се получава от другия. Намерете редакционното разстояние чрез динамично програмиране, като попълните динамичната таблица. Намерете и редакционното предписание, тоест една конкретна най-къса редица от операции, която води от едната дума до другата. Оцветете или щриховайте четливо клетките от таблицата, по които се съставя редакционното предписание.

Задача 3. Опишете на псевдокод алгоритъм с линейна времева сложност при най-лоши входни данни, който премахва ориентацията на ребрата на граф без кратни ребра и без примки, представен чрез списъци на съседствата. Върхове са числата от 1 до n вкл., Adj[u] е списъкът на ребрата, излизащи от u; Adj1[1…n] и Adj2[1…n] са масивите от списъци на входния и изходния граф съответно. Да се премахне ориентацията на ребрата, означава следното:

$$v \in \mathrm{Adj1}[u] \text{ или } u \in \mathrm{Adj1}[v] \Leftrightarrow v \in \mathrm{Adj2}[u] \Leftrightarrow u \in \mathrm{Adj2}[v].$$

Графът, получен на изхода, не трябва да съдържа кратни ребра и примки. С други думи, Adj2[u] не трябва да съдържа нито повторения, нито върха u. По условие Adj1[u] не съдържа повторения, нито върха u, обаче е възможно Adj1[u] да съдържа v и Adj1[v] да съдържа u (това не се смята за кратно ребро).

Задача 4. Намерете максималната и амортизираната времева сложност на алгоритъма NEG.

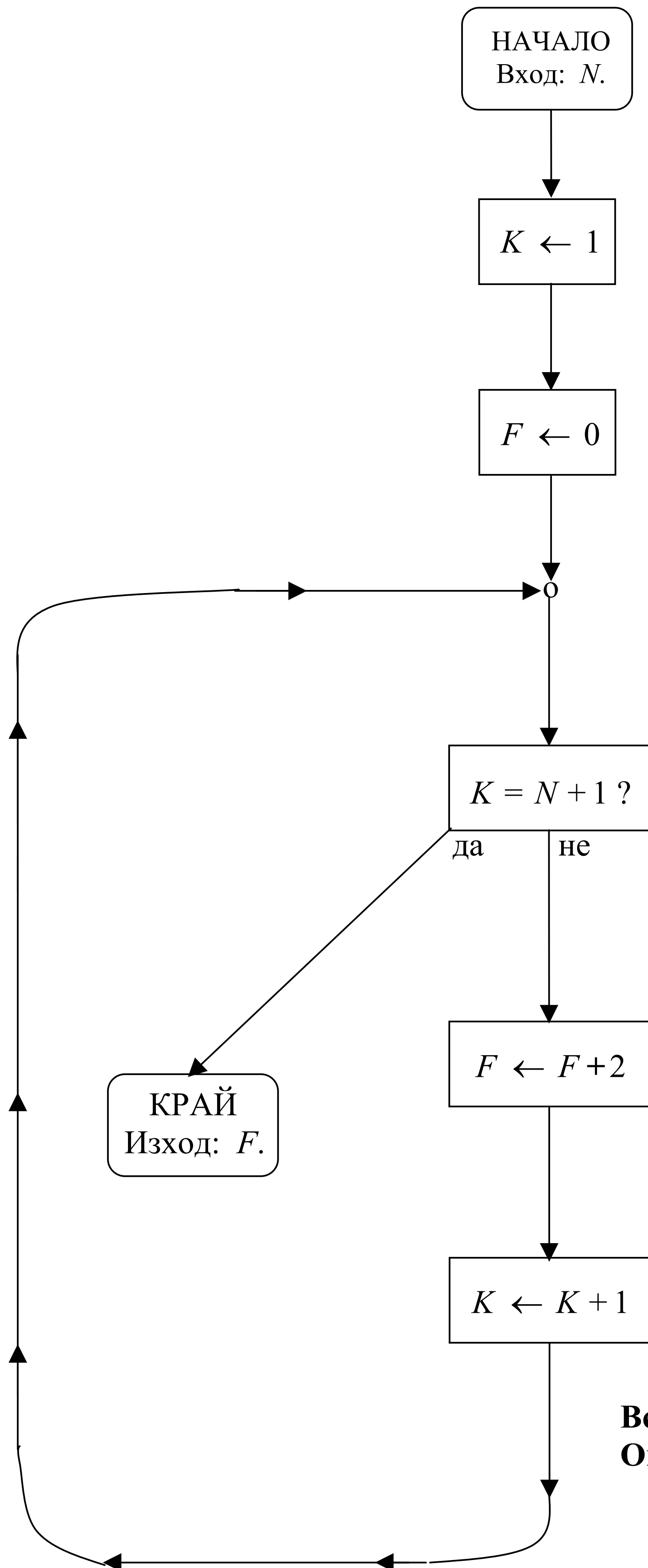
```
ALG(A[1...k]: числов масив)
1) for n ← 1 to k do
2)     NEG(A[1...n])

NEG(A[1...n]: числов масив)
1) if A[n] > 100
2)     p ← n
3)     while p ≥ 1 and A[p] > 0
4)         A[p] ← −A[p]
5)         p ← p − 1
```

Задача 5. Намерете очакваното време за изпълнение на алгоритъма.

```
PER(π[1...3n]: числов масив)
// пермутация без повторение
// на числата 1, 2, ... , 3n
1) repeat
2)     t ← rand(1 ... 3n)
3)     r ← rand(1 ... 3n)
4)     x ← π[t] mod 3
5)     y ← π[r] mod 3
6) until x ≠ y
7) print t, r
```

Задача 6. Какво връща следният алгоритъм? Отговорът да се обоснове с множество от инварианти. Приемете, че N е цяло неотрицателно число.



Всяка задача носи по една точка.
Оценката = 1 + точките.

РЕШЕНИЯ

Задача 1. Твърдението се доказва чрез редукция от задачата за спирането. Да допуснем, че съществува алгоритъм, проверяващ дали дадена променлива на дадена компютърна програма при даден вход ще приеме дадена стойност когато и да било по време на изпълнение на програмата. Чрез този алгоритъм решаваме задачата за спирането по следния начин:

```
Спира(програма P, входни данни D)
1) V ← име на променлива, което липсва в програмата P.
2) Добавяме оператора "V ← 0" в началото на програмата P.
3) Добавяме оператора "V ← 1" в края на програмата P
   и преди всяка команда за завършване на P.
4) return Променлива_получава_стойност(P,D,V,1).
```

Тук стрелката наляво е знак за присвояване на стойност, както е прието за алгоритми, описани на псевдокод.

Пример: Тази функция на Си заменя малките латински букви *a*, *b* и *c* със съответните им главни букви. Функцията обработва даден низ от знаци до неговия край (отбелязан с т. нар. нулев знак) или до срещането на знак, различен от споменатите три букви. Вляво е показана оригиналната функция, а вдясно — функцията, получена след добавянето на оператори за присвояване (оцветени в червено).

```
void P(char * D)
{
  char * S = D;
  while (*S != 0)
  {
    switch (*S)
    {
      case 'a': *S = 'A'; break;
      case 'b': *S = 'B'; break;
      case 'c': *S = 'C'; break;
      default: return;
    }
    ++S;
  }
}
```

```
void P(char * D)
{
  int V = 0;
  char * S = D;
  while (*S != 0)
  {
    switch (*S)
    {
      case 'a': *S = 'A'; break;
      case 'b': *S = 'B'; break;
      case 'c': *S = 'C'; break;
      default: V = 1; return;
    }
    ++S;
  }
  V = 1;
}
```

Ясно е, че ако пуснем променената програма P върху входните данни D, то нейната локална променлива V ще получи стойност 1 тогава и само тогава, когато променената програма P завършва изпълнението си; а това ще стане тогава и само тогава, когато първоначалната програма P завършва обработката на входните данни D. Това важи и в общия случай, т.е. редукцията е коректна.

Построената редукция показва, че ако е решима задачата дали променлива получава дадена стойност, то ще бъде решима също и задачата за спирането. Обаче знаем, че задачата за спирането е нерешима. Това противоречие показва, че направеното допускане не е вярно. Следователно алгоритмичната задача “Дали дадена променлива на дадена компютърна програма при даден вход ще приеме дадена стойност кога да е по време на изпълнение на програмата?” е нерешима.

Задача 2. Разстоянието на Левенщейн (тоест редакционното разстояние) между думите ДАГЕСТАН и АРЕСТАНТ е равно на 3, както се вижда от попълнената динамична таблица.

		А	Р	Е	С	Т	А	Н	Т
	0	1	2	3	4	5	6	7	8
Д	1	1	2	3	4	5	6	7	8
А	2	1	2	3	4	5	5	6	7
Г	3	2	2	3	4	5	6	6	7
Е	4	3	3	2	3	4	5	6	7
С	5	4	4	3	2	3	4	5	6
Т	6	5	5	4	3	2	3	4	5
А	7	6	6	5	4	3	2	3	4
Н	8	7	7	6	5	4	3	2	3

Оптималното решение се възстановява от долния десен към горния ляв ъгъл, при което се получава пътят от клетки, оцветени в жълто. Пътят се построява от края към началото, но в действителност се изминава от началото към края, тоест от горния ляв към долния десен ъгъл. Ако това движение се тълкува като преминаване от думата ДАГЕСТАН към думата АРЕСТАНТ, то:

— всяка стъпка надолу означава изтриване на буква;
— всяка стъпка надясно означава добавяне (вмъкване) на буква;
— всяка стъпка по диагонал надолу и надясно, при която числото нараства, означава замяна на буква.

Коя буква се изтрива, вмъква или заменя и коя е новата буква, се разбира от реда и стълба на съответната жълта клетка от таблицата.

Конкретно за двете дадени думи са достатъчни три операции, както личи от получения числен отговор (стойността в долната дясна клетка на таблицата). Редакционното предписание гласи:

1) Изтриваме буквата Д от думата ДАГЕСТАН и получаваме АГЕСТАН.
2) Заменяме буквата Г с буквата Р и получаваме думата АРЕСТАН.
3) Добавяме буквата Т в края и получаваме думата АРЕСТАНТ.

Задача 3. Нека n и m са съответно броят на върховете и броят на ребрата на дадения граф. Ако при обхождане на ребрата в списъците от масива Adj1 добавяме v към Adj2[u] и u към Adj2[v] за всяко $v \in$ Adj1[u], ще получим неориентиран мултиграф, ако $v \in$ Adj1[u] и $u \in$ Adj1[v] за някои върхове u и v. Ако проверяваме новите списъци преди всяко добавяне на ребро, ще избегнем тази опасност, но алгоритъмът ще има квадратична времева сложност.

Затова ще използваме матрица на съседствата за новия граф едновременно със списъците на съседства. С нейна помощ отбелязваме добавянето на ребро и проверяваме наличието на ребро за константно време (номерата на върховете, които са краища на реброто, са индексите на реда и стълба на онази клетка, чиято стойност трябва да зададем или прочетем). Матрицата на съседствата има размер $\Theta(n^2)$, но това се отнася до сложността по памет, не по време.

Няма нужда да четем цялата матрица: прочитаме само онези нейни клетки, които съответстват на ребрата на графа, а техният брой е $\Theta(m)$, т.е. четенето има линейна времева сложност.

Проблем създава само инициализирането на матрицата с нулеви стойности. То изисква време $\Theta(n^2)$, надхвърлящо ограничението от условието на задачата. Решението на проблема е да оставим матрицата неинициализирана. За целта използваме изучената структура от данни за работа с неинициализирана памет.

Описание на алгоритъма на псевдокод:

```
Премахване на ориентацията на ребрата на граф,
представен чрез списъци на съседствата(Adj1[1...n])
 1) Adj2[1...n]: масив от новите списъци на съседства;
 2) M[1...n][1...n]: матрица на съседствата на новия граф.
 3) // Оставяме матрицата M неинициализирана!
 4) for u ← 1 to n do
 5)    Adj2[u] ← празен списък
 6) for u ← 1 to n do
 7)    for v∈Adj1[u] do
 8)       if клетката M[u][v] е неинициализирана:
 9)          добавяме v към Adj2[u];
10)          добавяме u към Adj2[v];
11)          инициализираме M[u][v] и M[v][u] с единици.
12) return Adj2[1...n]
```

Използваната структура от данни за работа с неинициализирана памет не е декларирана явно, а само се подразбира. Тази структура се използва така: инициализира се на ред № 3, чете се на ред № 8, а се попълва на ред № 11. (На ред № 11 също се присвоява стойност 1 на две от клетките на матрицата.)

Всички неинициализирани клетки на матрицата на съседствата се тълкуват като нули, тоест означават, че между съответните два върха на новия граф все още няма ребро.

Предложеният алгоритъм има линейна времева сложност: първият цикъл изразходва време $\Theta(n)$, а двата вложени цикъла изразходват време $\Theta(n + m)$, защото всеки връх и всяко ребро се обхождат по веднъж.

Задача 4. Максималната времева сложност на алгоритъма NEG е $\Theta(n)$; тя се достига, когато всички елементи на входния масив са положителни числа и последното от тях е по-голямо от 100. Това са най-лошите входни данни, независимо дали става дума за алгоритъма ALG, или за подалгоритъма NEG.

Амортизираната сложност на подалгоритъма NEG може да се пресметне по агрегатния метод. Въпреки че едно отделно изпълнение на алгоритъма NEG може да бъде много дълго, не може да има много такива изпълнения. Наистина, при всяко такова изпълнение всички елементи на масива, обработени от NEG, стават отрицателни, а при следващи извиквания NEG не отива по-наляво от най-десния обработен елемент. Следователно върху всеки елемент на масива алгоритъмът NEG извършва поне една операция (сравнява го с числото 100) и най-много още две операции (сравнява елемента с нулата и сменя знака му). Така NEG извършва общо (за цялото изпълнение на главния алгоритъм ALG) между k и $3k$ операции. Тъй като ALG извиква NEG точно k пъти, то следва, че средният брой операции на едно извикване на NEG е между 1 и 3 вкл. Ето защо амортизираната времева сложност на подалгоритъма NEG е $\Theta(1)$.

Задача 5. В псевдокода се предполага, че извикването `rand(1 ... 3n)` връща случайно цяло число, равномерно разпределено от 1 до $3n$ включително, и че отделните извиквания са независими в съвкупност. Това важи еднакво за случайно избраните индекси на елементите и за стойностите на елементите. Тъй като от 1 до $3n$ вкл. има по n числа от всеки остатък при деление на 3, то има вероятност 2/3 двата остатъка x и y да са различни. Тази оценка е точна, а не приблизителна, тъй като изборът на индексите t и r е <u>със</u> връщане (тоест допуска се възможността $t = r$).

Времевата сложност на алгоритъма PER е от порядък броя на изпълненията на тялото на цикъла, а този брой е случайна величина, чието разпределение е геометрично $\mathrm{Ge}(p)$ от единицата — брой опити до първи успех включително. Ред № 6 от псевдокода на алгоритъма PER означава, че успех в тази задача е получаването на различни остатъци при деление на 3, откъдето следва, че вероятността за успех е $p = 2/3$.

От теорията на геометричното разпределение знаем, че $1/p = 3/2 = 1{,}5$ е математическото очакване на разглежданата случайна величина. Ето защо тялото на цикъла се изпълнява средно 1,5 пъти. Този брой не зависи от n, затова очакваното време за изпълнение на алгоритъма е от порядък $\Theta(1)$.

Задача 6. Алгоритъмът пресмята числото $2N$.