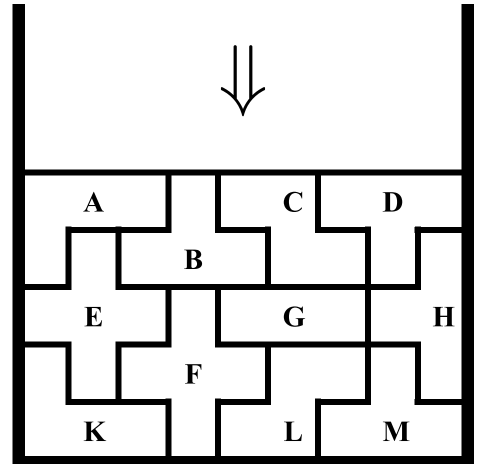
ИЗПИТ ПО "ДИЗАЙН И АНАЛИЗ НА АЛГОРИТМИ" — СУ, ФМИ, 07. 02. 2019 Г.

Имате право да използвате наготово всички алгоритми, изучени на лекции. Всеки алгоритъм, използван наготово, трябва да бъде посочен чрез името си на български език. Ако такъв алгоритъм има няколко реализации, трябва да се уточни използваната реализация. Когато използвате графи, описвайте подробно процеса на моделиране: какво представляват върховете и ребрата на графа. За алгоритми върху графи указвайте използваната алгоритмична схема (ако има такава) — обхождане в ширина или обхождане в дълбочина. Уточнявайте името на използваните сортировки.

Задача 1. Части от детска игра конструктор са прибрани в кутия. Частите се спускат в кутията една по една, отвесно — в посоката, указана от стрелката, без движения наляво и надясно. В какъв ред са прибрани фигурките? Достатъчно е да намерите една редица.



а) Решете задачата за случая, показан на картинката. **(2 точки)**
б) Моделирайте входните данни чрез граф. Опишете смисъла на върховете и ребрата в общия случай. **(5 точки)**
в) Съставете алгоритъм с линейна времева сложност, решаващ задачата по даден граф в общия случай. Опишете алгоритъма с думи. **(5 точки)**
г) Постройте графа Г за примера от картинката. **(3 точки)**
д) Изпълнете целия алгоритъм върху графа Г. **(5 точки)**

Задача 2. Алгоритъм за сортиране бил пуснат два пъти подред на един и същи компютър, обаче върху числови масиви с различни дължини. Първият масив съдържал един милион елемента и бил сортиран за 10 ms. Вторият масив съдържал три милиона елемента и бил сортиран за 90 ms. С кой от следните алгоритми се е случило това: сортиране чрез вмъкване, сортиране чрез сливане, пирамидално сортиране? Отговорът да се обоснове! **(10 точки)**
За кой от трите алгоритъма е най-вероятно да възникне недостиг на памет? **(10 точки)**

Задача 3. Съставете възможно най-бърз алгоритъм, който проверява дали даден масив $A[1...n]$ е пермутация без повторение на числата 1, 2, 3, ... , n. Опишете алгоритъма на псевдокод и анализирайте времевата сложност в най-лошия случай. **(20 точки)**

Задача 4. Има две купчинки: първата — с m, втората — с n камъчета Двама играчи се редуват да взимат камъчета. Който вземе последното камъче, печели. Всеки играч има две възможности:
— Да вземе произволен брой камъчета от едната купчинка (която си избере). Длъжен е да вземе поне едно камъче, но няма горна граница: ако желае, може да вземе цялата купчинка.
— Да вземе равен брой камъчета от двете купчинки. Длъжен е да вземе поне по едно камъче.
Опишете на псевдокод алгоритъм, който по дадени числа m и n за време $O((m+n)mn)$ определя кой играч има печеливша стратегия. Демонстрирайте алгоритъма при $m = n = 9$. **(20 точки)**
Опишете стратегията с думи или с псевдокод и я демонстрирайте при $m = n = 9$. **(20 точки)**

Задача 5. Дадена е квадратна двоична матрица $A[1...n][1...n]$ с n реда и n стълба. Индекса k (цяло число от 1 до n вкл.) наричаме приятен, ако ред № k съдържа само нули, а стълб № k — само единици (тези изисквания не се отнасят за елемента $A[k][k]$, който може да е произволен). По-формално, индексът k е приятен, ако и само ако $A[k][j] = 0$ за $\forall j \neq k$ и $A[i][k] = 1$ за $\forall i \neq k$. Търсим приятен индекс (ако съществува такъв). Докажете, че тази алгоритмична задача притежава времева сложност $\Theta(n)$ в най-лошия случай, като:
а) съставите алгоритъм с времева сложност $O(n)$ в най-лошия случай; **(10 точки)**
б) докажете долна граница на времевата сложност $\Omega(n)$ в най-лошия случай. **(10 точки)**

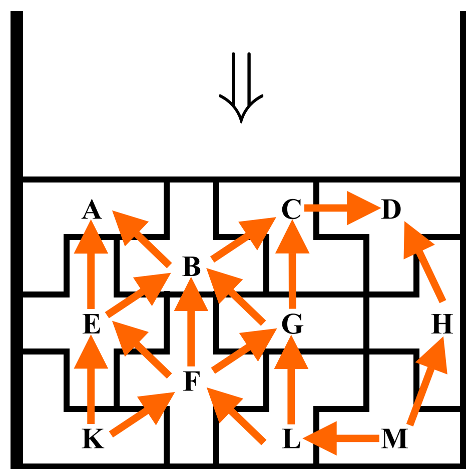
Р Е Ш Е Н И Я

Задача 1.

а) Една възможна подредба на показаните фигурки е следната: **M**, **L**, **K**, **F**, **E**, **G**, **H**, **B**, **C**, **D**, **A**.

б) В общия случай входните данни на задачата могат да бъдат описани чрез ориентиран граф:
- Върховете на графа съответстват на фигурките (частите от конструктора).
- Графът няма примки. Всяко ребро съответства на покриване на една фигурка от друга. По-точно, графът съдържа ребро от върха x към върха y тогава и само тогава, когато фигурката y покрива фигурката x, тоест контурите на двете фигурки притежават обща водоравна отсечка и фигурката y е над отсечката, а фигурката x е под отсечката.

в) По условие фигурките се спускат отвесно, без движения наляво и надясно. Следователно, ако една фигурка покрива друга, то покритата фигурка трябва да бъде сложена в кутията преди покриващата фигурка. Ребрата на графа сочат от покрита към покриваща фигурка, затова, ако подредим фигурките така, че покритите да са вляво от покриващите фигурки, то всички ребра ще сочат надясно. Такава подредба на върховете на графа се извършва с известния алгоритъм за ***топологично сортиране,*** използващ алгоритмичната схема ***обхождане в дълбочина.***



г) Графът за примера от условието е начертан на картинката вдясно: върховете на графа са обозначени с наименованията на фигурките, а ребрата са показани с оранжево.

д) Изпълняваме алгоритъма за топологично сортиране върху показания граф. Обхождаме графа в дълбочина, започвайки от произволен връх:

((((( ) (( )))) ( ))) (( ) ( ))

K F G B A A C D D C B G E E F K M L L H H M

Подреждаме върховете в ред, обратен на затварянето: **M**, **H**, **L**, **K**, **F**, **E**, **G**, **B**, **C**, **D**, **A**.
Това е един възможен ред на вкарване на фигурките в кутията.

Задача 2. Описаната случка е могла да стане единствено с алгоритъма сортиране чрез вмъкване: от изброените алгоритми само той има квадратична времева сложност (при лоши входни данни), останалите алгоритми имат времева сложност $n \log n$. До заключението за квадратична сложност достигаме, като съпоставим дължините на входните масиви с времената за сортирането им: на три пъти по-голям масив съответства девет пъти по-дълго време за обработка, което е признак за квадратична сложност.

Недостиг на памет с най-голяма вероятност ще възникне при алгоритъма сортиране чрез сливане: този алгоритъм е рекурсивен и прави копие на входните данни, затова изразходва много памет. Другите два алгоритъма са итеративни и сортират на място, тоест изискват съвсем малко памет, поради което при тях е твърде невероятно да възникне недостиг на памет.

Задача 3. Елементите на масива са цели числа и са малки в сравнение с дължината на масива. Затова можем да извършим проверката за линейно време, като използваме идеята на алгоритъма ***сортиране чрез броене.*** Ще използваме булеви флагове вместо броячи.

```
CheckPermutation(A[1...n])
C[1...n]: array of boolean // флагове
for k ← 1 to n do
   C[k] ← false
for k ← 1 to n do
   if (A[k] is not an integer) or (A[k] < 1) or (A[k] > n)
      return false // Намерена е недопустима стойност.
   if C[A[k]]
      return false // Намерено е повторение.
   C[A[k]] ← true
for k ← 1 to n do
   if not C[k]
      return false // Липсваща стойност.
return true // Масивът A е пермутация без повторение на 1, 2, ... , n.
```

Времевата сложност на всеки от трите цикъла е $\Theta(n)$, следователно $\Theta(n)$ е времевата сложност и на целия алгоритъм. Тази сложност важи за всякакви входни данни. Тя е с оптимален порядък: всеки алгоритъм, който решава задачата, трябва да прочете всяко число от входните данни, защото всеки елемент на масива може да нарушава някое изискване (тривиална долна граница по дължината на входа).

Можем да спестим около една трета от времето на алгоритъма, като премахнем последния цикъл. Проверката за липсваща стойност е излишна: след като сме установили, че дадените n числа са измежду числата 1, 2, 3, ... , n и никое не се повтаря, то не е възможно и да липсва някое. Тази оптимизация носи допълнителни 10 точки.

Изложеният алгоритъм е оптимален по време, но не и по памет: той използва допълнителна памет от порядък $\Theta(n)$ за масива $C[1...n]$. Можем да спестим тази памет, ако е разрешено да променяме входните данни: ролята на флаг ще играе знаковият бит на всеки от елементите на масива A.

```
CheckPermutation(A[1...n])
for k ← 1 to n do
   if (A[k] is not an integer) or (A[k] < 1) or (A[k] > n)
      return false // Намерена е недопустима стойност.
for k ← 1 to n do
   if A[abs(A[k])] < 0
      return false // Намерено е повторение.
   A[abs(A[k])] ← -A[abs(A[k])]
return true // Масивът A е пермутация без повторение на 1, 2, ... , n.
```

Този вариант на алгоритъма има сложност по време $\Theta(n)$ и сложност по памет $\Theta(1)$. С други думи, той е оптимален не само по време, но и по памет, поради което носи допълнителни 20 точки (дори ако третият цикъл не е премахнат). Бонусите за двете оптимизации се събират, тоест последният вариант (с използване на знаковия бит като флаг и без проверка за липсваща стойност) носи допълнителни 30 точки (20 точки за едната оптимизация и 10 точки за другата).

Задача 4 може да се реши за време $O((m+n)mn)$ чрез ***динамично програмиране.*** Наредената двойка от числа (m, n) — брой камъчета във всяка купчинка — ще наричаме позиция. Казваме, че една позиция е печеливша, ако играчът, който е на ход в тази позиция, има печеливша стратегия. Една позиция е печеливша точно тогава, когато има ход, който води от нея до губеща позиция. Една позиция е губеща точно тогава, когато всички ходове от нея водят до печеливши позиции. Ако началната позиция (m, n) е печеливша, то следва, че първият играч има печеливша стратегия. В противен случай, т.е. ако началната позиция е губеща, вторият играч има печеливша стратегия.

```
Winning(m, n)  // Връща истина (true) ⇔ позицията (m, n) е печеливша.
dyn[0...m][0...n]: array of boolean    // Печеливши и губещи позиции.
heap1[0...m][0...n]: array of integers // Печеливш ход: брой камъчета,
heap2[0...m][0...n]: array of integers // взети от всяка купчинка.
for i ← 0 to m do
   for j ← 0 to n do
      dyn[i][j] ← false
      heap1[i][j] ← 0
      heap2[i][j] ← 0
      for k ← 0 to i-1 do
         if dyn[k][j] = false
            dyn[i][j] ← true
            heap1[i][j] ← i-k // Взимаме i-k камъчета от купчинка № 1.
            goto nextJ
      for k ← 0 to j-1 do
         if dyn[i][k] = false
            dyn[i][j] ← true
            heap2[i][j] ← j-k // Взимаме j-k камъчета от купчинка № 2.
            goto nextJ
      for k ← min(i,j) downto 1 do
         if dyn[i-k][j-k] = false
            dyn[i][j] ← true
            heap1[i][j] ← k
            heap2[i][j] ← k  // Взимаме k камъчета от всяка купчинка.
            goto nextJ
      nextJ: do nothing  // Празен оператор: край на цикъла с брояч j.
return dyn[m][n], heap1, heap2
```

Демонстрация на алгоритъма при $m = n = 9$ е показана в таблицата вдясно. Измежду стоте анализирани позиции има само седем губещи; другите позиции са печеливши. За всяка печеливша позиция се пази и ходът, който води до победа (не е показан в таблицата). Ходът представлява движение от текущата клетка (тоест печеливша позиция) до губеща позиция наляво по реда, надолу по стълба или наляво и надолу по диагонал. Например позицията $(9; 9)$ е печеливша: първият играч печели, взимайки 9 камъчета от всяка купчинка; по този начин противникът му остава в губещата позиция $(0; 0)$.

m \ dyn	0	1	2	3	4	5	6	7	8	9
9	T	T	T	T	T	T	T	T	T	T
8	T	T	T	T	T	T	T	T	T	T
7	T	T	T	T	F	T	T	T	T	T
6	T	T	T	T	T	T	T	T	T	T
5	T	T	T	F	T	T	T	T	T	T
4	T	T	T	T	T	T	T	F	T	T
3	T	T	T	T	T	F	T	T	T	T
2	T	F	T	T	T	T	T	T	T	T
1	T	T	F	T	T	T	T	T	T	T
0	F	T	T	T	T	T	T	T	T	T

n

Задача 5.

а) Съществува най-много един приятен индекс. Той може да бъде открит за време $O(n)$ така:

```
isPleasant(A[1...n][1...n], k)
for j ← 1 to n do
   if (j ≠ k) and (A[k][j] = 1 or A[j][k] = 0)
      return false
return true

findPleasant(A[1...n][1...n]) // връща -1, ако няма приятен индекс
k ← 1
for j ← 2 to n do
   if A[k][j] = 1   // индексът k със сигурност не е приятен
      k ← j
if isPleasant(A[1...n][1...n], k)
   return k
return -1
```

б) Долната граница се доказва на два етапа. Първо, няма реализация на функцията `isPleasant` с по-малко от $2(n-1)$ прочитания на стойности на елементи на матрицата A: всяка реализация трябва да прочете ред № k и стълб № k (без клетката, където се пресичат), за да провери изискването за приятност на k, иначе няма как да научи стойностите на тези $2(n-1)$ елемента, защото елементите на матрицата са независими. И така, алгоритмичната задача `isPleasant` изисква време $\Omega(n)$.

Второ, следната редукция показва, че задачата `findPleasant` също изисква време $\Omega(n)$.

```
isPleasant(A[1...n][1...n], k)
return findPleasant(A[1...n][1...n]) = k
```

Самата редукция изразходва константно време (за сравняване на върнатата стойност с k), тоест редукцията е достатъчно бърза за целите на доказателството. Коректността на редукцията следва от единствеността на приятния индекс (когато съществува такъв).