Зад. 1

Дадени са $k \in \mathbb{N}^+$ и $A[1 \,..\, n] \in \mathbb{Z}^n,\ n \in \mathbb{N}^+$. Да се намеи $\max\{L \mid \exists\, i \in \{1, \ldots, n-k+1\}\ \forall\, j \in \{i, \ldots, i+k-1\}$ е изп. $A[j] \geq L\}$.

Пример:

$$\begin{cases} \text{Вход} : A[1\,..9] = [2,\ 5,\ 8,\ 4,\ 1,\ 6,\ 7,\ 8,\ -1],\ k = 3 \\ \text{Изход} : 6 \end{cases}$$

Обяснение на пример:

Търсим $k = 3$ поредни числа от масива, такива че минималното от тях да е максимално голямо.. Тоест все едно правим $\max\{\min\{2,\ 5,\ 8\},\ \min\{5,\ 8,\ 4\},\ \min\{8,\ 4,\ 1\},\ \ldots,\ \min\{7,\ 8,\ -1\}\} = \max\{2,\ 4,\ 1,\ 1,\ 1,\ 6,\ -1\} = 6$

Идея (наивна)

За всеки подмасив $A[i\,..i+k-1]$ намираме за линейно време минимума и взимаме максималния такъв.. общо време $\theta(k(n-k+1)) \overset{k \leq n}{\leq} \theta(kn)$.

Идея (средно добра)

Двоично по отговора. Времевата сложност ще бъде $\theta(\log(\max(A) - \min(A))\, n)$.

Идея (средно добра)

Самобалансиращо се наредено дърво. Времевата сложност ще бъде $\theta(\log(k)\, n)$.

Идея (много добра)

Използвайки deque. Остава са самостоятелна работа. Времевата сложност ще бъде $\theta(n)$.

Идея (много добра)

Ще вкараме в опашка, поддържаща бонус операция min() с $\theta(1)$ амортизирана сложност, първите k елемента. Ще имаме променлива, която ни държи максималния минимум. Ще изкарваме един елемент от опашката и ще добавяме нов и ще проверяваме дали новия минимум не е по-голям от максималния минимум до момента.

Обяснение на идеята:

Ще използваме примера отгоре. Започваме като запълваме опашката с първите $k = 3$ елемента от масива.

2 5 8 ,min() $\longmapsto$ 2

5 8 4 ,min() $\longmapsto$ 4

8 4 1 ,min() $\longmapsto$ 1

4 1 6 ,min() $\longmapsto$ 1

1 6 7 ,min() $\longmapsto$ 1

6 7 8 ,min() $\longmapsto$ 6

7 8 −1 ,min() $\longmapsto$ −1

Взимаме максималния такъв минимум и сме готови. Преди да преминем към реализацията на опашка с добавена такава операция, нека първо реализираме стек с бонус такава операция. Опашката ще реализираме посредством 2 стека.

Стек, поддържащ операцията min() за константно време и линейна допълнителна памет

Ще имаме два стека s и m : s е нормален а, m е със св-вото: $m.\text{top}() = \min(s)$. Нека разгледаме пример:

$$\text{push}(5) \longmapsto \begin{cases} s : & |\,5 \\ m : & |\,5 \end{cases}$$

$$\text{push}(6) \mapsto \begin{cases} s: & \boxed{|\,5\ 6} \\ m: & \boxed{|\,5\ 5} \end{cases}$$

$$\text{push}(3) \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 3} \\ m: & \boxed{|\,5\ 5\ 3} \end{cases}$$

$$\text{push}(4) \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 3\ 4} \\ m: & \boxed{|\,5\ 5\ 3\ 3} \end{cases}$$

$$\text{push}(5) \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 3\ 4\ 5} \\ m: & \boxed{|\,5\ 5\ 3\ 3\ 3} \end{cases}$$

$$\text{push}(1) \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 3\ 4\ 5\ 1} \\ m: & \boxed{|\,5\ 5\ 3\ 3\ 3\ 1} \end{cases}$$

…

Ясно е, че pop() би извилако s.pop() & m.pop().

Ясно е, че top() би извикало s.top().

Ясно е, че min() би извикало m.top().

Реализацията е оставена за упражнение.

Стек, поддържащ операцията min() за константно време и константа допълнителна памет

Тук правим комбинация на горните 2 стека.. Имаме само 1 стек, който нито съдържа минималните елементи, нито съдържа истинските елементи, ами някаква комбинация на двете..

Ще са ни необходими един обикновен стек s и една бонус променлива m, която ще ни съдържа текущия минимум. Ако добавяме елемент a, който е по-голям или равен на текущия минимум m, то добавяме елемента a без промяна. Ако ли не, то добавяме в стека $(2\,a - m)$ и актуализираме минимума m. Използвайки тази магическа формула ще можем да разберем кой е бил предходния минимум при pop()-ване на текущия и по-конкретно: $\textit{стария минимум} = (2\,m - s.\text{top}())$. Съответно top() = max($s$.top(), m) Нека разгледаме пример:

$$\text{push}(5) \mapsto \begin{cases} s: & \boxed{|\,5} \\ m: & 5 \end{cases}; \text{top}() \mapsto \max(5,\ 5) = 5;$$

$$\text{push}(6) \mapsto \begin{cases} s: & \boxed{|\,5\ 6} \\ m: & 5 \end{cases}; \text{top}() \mapsto \max(6,\ 5) = 6;$$

$$\text{push}(3) \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 1} \\ m: & 3 \end{cases}; \text{top}() \mapsto \max(1,\ 3) = 3;$$

$$\text{push}(4) \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 1\ 4} \\ m: & 3 \end{cases}; \text{top}() \mapsto \max(4,\ 3) = 4;$$

$$\text{push}(5) \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 1\ 4\ 5} \\ m: & 3 \end{cases}; \text{top}() \mapsto \max(5,\ 3) = 5;$$

$$\text{push}(1) \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 3\ 1\ 5\ -1} \\ m: & 1 \end{cases}; \text{top}() \mapsto \max(-1,\ 1) = 1;$$

$$\text{pop}() \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 1\ 4\ 5} \\ m: & 3 \end{cases}; \text{top}() \mapsto \max(5,\ 3) = 5;$$

$$\text{pop}() \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 1\ 4} \\ m: & 3 \end{cases}; \text{top}() \mapsto \max(4,\ 3) = 4;$$

$$\text{pop}() \mapsto \begin{cases} s: & \boxed{|\,5\ 6\ 1} \\ m: & 3 \end{cases}; \text{top}() \mapsto \max(1,\ 3) = 3;$$

$$\text{pop}() \mapsto \begin{cases} s: & \boxed{|\,5\ 6} \\ m: & 5 \end{cases}; \text{top}() \mapsto \max(6,\ 5) = 6;$$

$$\text{pop}() \mapsto \begin{cases} s: & \boxed{|\,5} \\ m: & 5 \end{cases}; \text{top}() \mapsto \max(5,\ 5) = 5;$$

Примерна реализация:

```
1. Struct StackWithMinInplace :
2.      s ← Stack.Init()
3.      min ← NULL
4.      push (a)
5.      pop ()
6.      top ()
7.      min ()
8.      isEmpty ()
```

```
1. StackWithMinInplace.push (a) :
2.      if s.isEmpty() = TRUE then
3.              s.push(a)
4.              min ← a
5.      if a ≥ min then
6.              s.push (a)
7.      else
8.              s.push(2 * a − min)
9.              min ← a
```

```
1. StackWithMinInplace.pop () :
2.      if s.top() ≥ min then
3.              return s.pop()
4.      t ← min
5.      min ← 2 * min − s.top()
6.      s.pop()
7.      return t
```

```
1. StackWithMinInplace.top() :
2.      return max(s.top(), min)
```

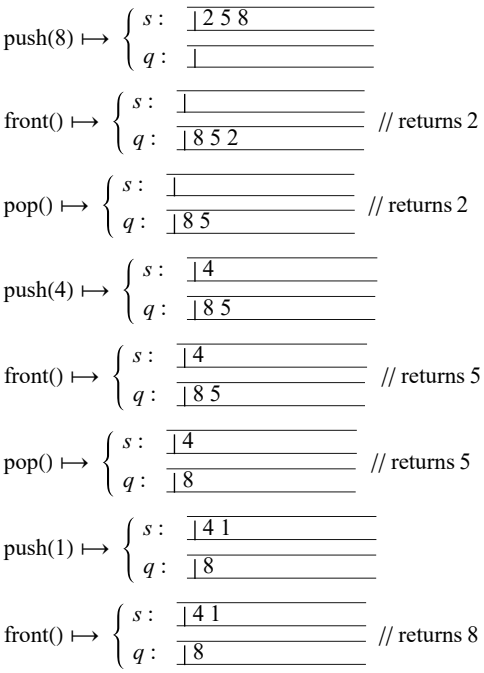
```
1. StackWithMinInplace.min() :
2.      return min
```

```
1. StackWithMinInplace.isEmpty() :
2.      return s.isEmpty()
```

Нека сега разгледаме как бихме реализирали опашка посредством 2 стека. Ще са ни необходими 2 стека s и q. Първия от които със семантика на стек, а другия на опашка (той пак си е стек, но елементите са подредени в правилния ред). Когато добавяме елемент a **винаги** го добавяме най-отгоре на s, т.е викаме s.push(a). Когато викаме front() или pop() имаме 2 случая - q.isEmpty() = TRUE и q.isEmpty() = FALSE. При първия случай това което правим е че прехвърляме всички елементи от s във q, като им обръщаме реда. След това викаме съответно q.top() или q.pop(). Когато q.isEmpty() = FALSE, то директно викаме q.top() или q.pop(). Нека разгледаме пример:

push(2) $\longmapsto$ $\begin{cases} s: & \overline{\vert\, 2} \\ q: & \overline{\vert} \end{cases}$

push(5) $\longmapsto$ $\begin{cases} s: & \overline{\vert\, 2\ 5} \\ q: & \overline{\vert} \end{cases}$



…

Ясно е, че ако реализираме опашка по описания по-горе начин, но използвайки StackWithMinInplace вместо обикновен стек, то можем да поддържаме операцията минимум за опашката по следния начин: min = min(*s*.min(), *q*.min()).. разбира се трябва да вземем в предвид случая, когато (*q*.isEmpty() ∨ *s*.isEmpty) = TRUE.

Примерна реализация:

```
1. Struct QueueWithMinInplace :
2.      s ← StackWithMinInplace.Init()
3.      q ← StackWithMinInplace.Init()
4.      push (a)
5.      pop ()
6.      front ()
7.      min ()
8.      isEmpty ()
```

```
1. QueueWithMinInplace.push(a) :
2.      s.push(a)
```

```
1. QueueWithMinInplace.pop() :
2.      if q.isEmpty() = TRUE then
3.              while s.isEmpty() = FALSE do
4.                      q.push(s.pop())
5.      return q.pop()
```

```
1. QueueWithMinInplace.front() :
2.      if q.isEmpty() = TRUE then
3.              while s.isEmpty() = FALSE do
4.                      q.push(s.pop())
5.      return q.top()
```

```
1. QueueWithMinInplace.min() :
2.       if q.isEmpty() = FALSE and s.isEmpty() = FALSE then
3.                return min(q.min(), s.min())
4.       else if q.isEmpty() = FALSE then
5.                retrun q.min()
6.       return s.min()
```

```
1. QueueWithMinInplace.isEmpty() :
2.       return q.isEmpty() and s.isEmpty()
```

Сложност на операциите:

· push(a) е $\theta(1)$

.pop() е $O(n)$, където n е броя елементи в s .. но това е най − лошия случай .. в почти всички случаи ще е де факто $O(1)$ .. оттук излиза амортизирана сложност $\theta(1)$

· front() е аналогично на pop()

· min() е $\theta(1)$

· isEmpty() е $\theta(1)$

Забележка В текушия курс не разглеждаме официално какво е амортизирана сложност, затова при използване на изложената по-горе структура от данни в конкретен алгоритъм не можем да използваме наготово, че има амортизирана $\theta(1)$ сложност.. ще трябва за конкретен алгоритъм да правим подходящи разсъждения.

Нека се върнем на задачата и ѝ направим реализация:

```
1. Task1(A[1 .. n], k) : // A ∈ ℤ^n, n ∈ ℕ^+, k ∈ ℕ^+
2.       q ← QueueWithMinInplace.Init()
3.       for i ← 1 to k
4.                q.push(A[i])
5.       L ← q.min()
6.       for i ← k + 1 to n
7.                q.pop()
8.                q.push(A[i])
9.                L ← max(L, q.min())
10.      return L
```

Сложност Както споменахме по-горе в **Забележка**-та, не можем да използваме, че q.pop() е с амортизиране $\theta(1)$ сложност. Нека обаче се убедим, че сложността на алгоритъма по-горе е баш (т.е не в амортизиран смисъл) $\theta(n)$. Ето ги и разсъжденията:

Всеки елемент от масива го добавяме най-много един път в опашката. Аналогично всеки елемент от масива го премахваме най-много един път от масива. Ако разгледаме какво се случва в опашката, разгледана като 2 стека, то всеки елемент бива довабян и изваждан най-много по 1 път във всеки стек.. тоест всеки елемент бива добавян и изваждан най-много 4 пъти (вкарване в *стек1*, изваждане от *стек1*, вкарване в *стек2*, изваждане от *стек2*). Тоест имаме $\Omega(4\,n)$ работа. Имаме тривиална долна граница $O(n)$, откъдето получихме $\theta(n)$. Това разсъждение беше за редове $3 \div 4$, $6 \div 8$. Лесно се вижда, че редове 2, 5, (6 със 8), 10 са ни със сложност $O(n)$, откъдето заключихме, че сложността на целия алгоритъм е $\theta(n)$.

Деф Мажоранта на масив $A[1 \,..\, n]$ наричаме елемент $m \in A[1 \,..\, n]$, който се среща повече (строго) от половината пъти, или казано по-друг начин се среща $\geq \left\lfloor \frac{n}{2} \right\rfloor + 1$ пъти.

Зад. 1

Даден е масив $A[1 .. n] \in \mathbb{N}^n$, $n \in \mathbb{N}^+$. Да се състави и докаже формално максимално бърз алгоритъм, намиращ мажорантата на $A[1 .. n]$, при условие че:

a) ни е гарантирано, че в масива има мажоранта

b) не ни е гарантирано, че в масива има мажоранта

a)

```
1. MajorityElementGuarantee(A[1 .. n]) : A ∈ ℕ^n, n ∈ ℕ^+, A има мажоранта
2.      curr ← A[1]
3.      cnt ← 1
4.      for i ← 2 to n
5.              if curr = A[i] then
6.                      cnt ← cnt + 1
7.              else
8.                      cnt ← cnt − 1
9.                      if cnt = 0 then
10.                             curr ← A[i]
11.                             cnt ← 1
12.     return curr
```

Горния алоритъм не е никак тривиално да се докаже, че е коректен. Нека се опитаме да го формализираме:

Идея Ще *чифтосаме (make pair) максимално* много двойки различни елементи (т.е със различни стойности). След като сме *чифтосали* максимално много елементи, то ще имаме поне 1 *нечифтосан* елемент. Този елемент ще ни е мажорантата.

Лема Ако масива $A[1 .. n]$ има мажоранта, то за всички възможни *максимални чифтосвания* имаме, че елемента мажоранта ще бъде поне един път *нечифтосан*.

Док. (Лема)

Нека мажорантата я означим с maj, а броя срещания на maj в $A[1 .. n]$ с m. Допускаме противното.

Ясно е, че няма как да *чифтосаме* maj със maj.. тоест трябва да го *чифтосаме* с някой друг елемент от масива $A[1 .. n]$. Тогава имаме поне $2m \geq 2\left\lfloor \frac{n}{2} \right\rfloor + 2$ елемента.

1 сл. ($n \equiv 0 \pmod 2$)

$2\left\lfloor \frac{n}{2} \right\rfloor + 2 = n + 2$.. противоречие - имаме n елемента.

2 сл. ($n \equiv 1 \pmod 2$)

$2\left\lfloor \frac{n}{2} \right\rfloor + 2 = n + 1$.. противоречие - имаме n елемента.

Следователно имаме поне една *нечифтосана* мажоранта.

Инвариант: При всяко k-то достигане на ред 4 имаме, че
$i = k + 1$ & единствения елемент, който не е бил *чифтосан* в подмасива $A[1 .. i - 1]$ е curr, и то cnt на брой пъти *нечифтосан*.

Заб Има се предвид, че се правят *чифтове* само между елементите от подмасива $A[1 .. i - 1]$.. т.е всички елементи от $A[i .. n]$ са *нечифтосани*.

База: $k = 1$

Тук $i = 1 + 1 = 2$. Също така единствения *нечифтосан* елемент в $A[1\ ..2-1]$ е curr = $A[1]$ и то cnt = 1 на брой пъти *нечифтосан*.

Поддръжка:

Нека е изпълнено за някое непоследно k-то достигане на ред 5. Ще докажем за $(k+1)$-во достигане на ред 5.

Ясно е, че $i_{new} = i_{old} + 1$.

1 сл. ($curr_{old} = A[i_{old}]$)

Тогава ($curr_{new} = curr_{old}$) & ($cnt_{new} = cnt_{old} + 1$).

От ИП имаме, че единствения елемент, който не е бил *чифтосан* в подмасива $A[1\ ..i_{old}-1]$ е $curr_{old}$ и то cnt_{old} на брой пъти.

Тоест имаме, че единствения елемент, който не е бил *чифтосан* в подмасива $A[1\ ..i_{new}-2]$ е $curr_{new}$ и то $cnt_{new} - 1$ на брой пъти.

Но $curr_{old} = A[i_{old}]$, т.е. $curr_{new} = A[i_{new}-1]$.

Оттук имаме, че единствения елемент, който не е бил *чифтосан* в подмасива $A[1\ ..i_{new}-1]$ е $curr_{new}$ и то cnt_{new} на брой пъти.

2.1 сл. ($curr_{old} \neq A[i_{old}]$ & $cnt_{old} > 1$)

Тогава ($curr_{new} = curr_{old}$) & ($cnt_{new} = cnt_{old} - 1$).

От ИП имаме, че единствения елемент, който не е бил *чифтосан* в подмасива $A[1\ ..i_{old}-1]$ е $curr_{old}$ и то $cnt_{old} > 1$ на брой пъти.

Тоест имаме, че единствения елемент, който не е бил чифтосан в подмасива $A[1\ ..i_{new}-2]$ е $curr_{new}$ и то $cnt_{new} + 1$ на брой пъти.

Но $curr_{old} \neq A[i_{old}]$, т.е. $curr_{new} \neq A[i_{new}-1]$. Него елемент го *чифтосваме* с някой от cnt_{old}-те на брой елемента $curr_{old}$.

Оттук имаме, че единствения елемент, който не е бил *чифтосан* в подмасива $A[1\ ..i_{new}-1]$ е $curr_{new}$ и то cnt_{new} на брой пъти.

2.2 сл. ($curr_{old} \neq A[i_{old}]$) & ($cnt_{old} = 1$)

Тогава ($curr_{new} = A[i_{old}]$) & ($cnt_{new} = 1$).

От ИП имаме, че единствения елемент, който не е бил *чифтосан* в подмасива $A[1\ ..i_{old}-1]$ е $curr_{old}$ и то $cnt_{old} = 1$ на брой пъти.

Но $curr_{old} \neq A[i_{old}]$, т.е $curr_{old} \neq A[i_{new}-1]$. *Чифтосваме* $curr_{old}$ със $A[i_{new}-1]$.

Вече всички елементи са ни чифтосани в подмасива $A[1\ ..i_{new}-1]$ са 0, но според нашата програма те трябва да са 1.. **не сработи**.

Нека разгледаме следната преработка на псевдокода:

```
1. MajorityElementGuarantee(A[1 .. n]) : A ∈ ℕ^n, n ∈ ℕ^+, A има мажоранта
2.     curr ← A[1]
3.     cnt ← 1
4.     f ← 0
5.     for i ← 2 to n
6.         if curr = A[i] then
7.             cnt ← cnt + 1
8.         else
9.             cnt ← cnt − 1
10.            if cnt = 0 then
11.                curr ← A[i]
12.                cnt ← 1
13.                f ← 1 − f
14.    return curr
```

Този псевдокод очевидно е еквивалентен на оригиналния.. единствено добавихме нова променлива f със семантика на флаг, която не бива използвана никъде.. само ѝ се присвояват стойности.

Сега с тази допълнителна информация можем да формулираме наново инвариант и да се надяваме, че информацията ще ни бъде достатъчна.

Инвариант: При всяко k-то достигане на ред 5 имаме, че
$i = k + 1$ & $f \in \{0, 1\}$ & единствения елемент, който не е бил чифтосан в подмасива $A[1 .. i - 1]$ е curr, и то cnt $- f$ на брой пъти.

База: $k = 1$

Тук $i = 1 + 1 = 2$ & $f = 0 \in \{0, 1\}$. Също така единствения *нечифтосан* елемент в $A[1 .. 2 - 1]$ е curr $= A[1]$ и то cnt $- f = 1$ на брой пъти.

Поддръжка:

Нека е изпълнено за някое непоследно k-то достигане на ред 5. Ще докажем за $(k + 1)$-во достигане на ред 5.

Ясно е, че $i_{new} = i_{old} + 1$.

Ясно е, че $(f_{new} = f_{old}) \vee (f_{new} = 1 - f_{old}) \overset{f_{old} \in \{0,1\}}{\Rightarrow} f_{new} \in \{0, 1\}$.

1 сл. $(curr_{old} = A[i_{old}])$

Тогава $(curr_{new} = curr_{old})$ & $(cnt_{new} = cnt_{old} + 1)$ & $(f_{new} = f_{old})$.

От ИП имаме, че единствения елемент, който не е бил *чифтосан* в подмасива $A[1 .. i_{old} - 1]$ е $curr_{old}$ и то $cnt_{old} - f_{old}$ на брой пъти.

Тоест имаме, че единствения елемент, който не е бил *чифтосан* в подмасива $A[1 .. i_{new} - 2]$ е $curr_{new}$ и то $cnt_{new} - 1 - f_{new}$ пъти.

Но $curr_{old} = A[i_{old}]$, т.е. $curr_{new} = A[i_{new} - 1]$.

Оттук имаме, че единствения елемент, който не е бил *чифтосан* в подмасива $A[1 .. i_{new} - 1]$ е $curr_{new}$ и то $cnt_{new} - f_{new}$ на брой пъти.

2.1 сл. $(curr_{old} \neq A[i_{old}]$ & $cnt_{old} > 1)$

Тогава $(curr_{new} = curr_{old})$ & $(cnt_{new} = cnt_{old} - 1)$ & $(f_{new} = f_{old})$.

От ИП имаме, че
единствения елемент, който не е бил *чифтосан* в подмасива $A[1 .. i_{old} - 1]$ е $curr_{old}$ и то $cnt_{old} - f_{old} \geq 1$ на брой пъти.

Тоест имаме, че
единствения елемент, който не е бил *чифтосан* в подмасива $A[1 .. i_{new} - 2]$ е $curr_{new}$ и то $cnt_{new} - f_{new} + 1$ на бр. пъти.

Но $curr_{old} \neq A[i_{old}]$, т.е. $curr_{new} \neq A[i_{new} - 1]$. Него елемент го *чифтосваме* с някой от $(cnt_{old} - f_{old})$-те на брой елемента $curr_{old}$.

Оттук имаме, че
единствения елемент, който не е бил *чифтосан* в подмасива $A[1 .. i_{new} - 1]$ е $curr_{new}$ и то $cnt_{new} - f_{new}$ на брой пъти (потенциално 0).

2.2.1 сл. $(curr_{old} \neq A[i_{old}]$ & $cnt_{old} = 1$ & $f_{old} = 0)$

Тогава $(curr_{new} = A[i_{old}])$ & $(cnt_{new} = 1)$ & $(f_{new} = 1)$.

От ИП имаме, че
единствения елемент, който не е бил *чифтосан* в подмасива $A[1 .. i_{old} - 1]$ е $curr_{old}$ и то $cnt_{old} - f_{old} = 1$ на брой пъти.

Но $curr_{old} \neq A[i_{old}]$, т.е $curr_{old} \neq A[i_{new} - 1]$. Него елемент го *чифтосваме* с единствения нечифтосан $curr_{old}$.

Вече всички елементи са ни *чифтосани* в подмасива $A[1 .. i_{new} - 1]$. Тоест е вярно, че
единствения елемент, който не е бил чифтосан в подмасива $A[1 .. i_{new} - 1]$ е $curr_{new}$ и то $cnt_{new} - f_{new} = 0$ на брой пъти.

2.2.2 сл. $(curr_{old} \neq A[i_{old}]$ & $cnt_{old} = 1$ & $f_{old} = 1)$

Тогава $(curr_{new} = A[i_{old}])$ & $(cnt_{new} = 1)$ & $(f_{new} = 0)$.

От ИП имаме, че
единствения елемент, който не е бил *чифтосан* в подмасива $A[1 .. i_{old} - 1]$ е $curr_{old}$ и то $cnt_{old} - f_{old} = 0$ на брой пъти. С други думи

от ИП знаем, че всички елементи са *чифтосани*.

Тогава елемента

$A[\underbrace{i_{\text{old}}}_{i_{\text{new}}-1}] = \text{curr}_{\text{new}}$ е единствения елемент, който не е бил *чифтосан* в подмасива $A\left[1 \,..\, \underbrace{i_{\text{old}}}_{i_{\text{new}}-1}\right]$ и то $\text{cnt}_{\text{new}} - f_{\text{new}} = 1$ на брой пъти.

Терминация:

При $k = n$-то достигане на ред 5 имаме $i = k + 1 = n + 1$ ще се прекрати изпълнението на цикъла. Тогава връща curr на ред 14. Тоест имаме, че единствения елемент, който не е бил *чифтосан* в подмасива $A\left[1 \,..\underbrace{i-1}_{n}\right]$ и то $\text{cnt} - f$ на брой пъти е curr.

1 сл. ($\text{cnt} - f = 0$)

Този случай е невъзможно да настъпи тъй като според **Лема-**та ни е гарантирано, че ще имаме поне 1 *нечифтосана* мажоранта.

2 сл. ($\text{cnt} - f > 0$)

От **Лема**-та знаем, че мажорантата ни е гарантирано, че ще я имаме поне 1 път *нечифтосана*. От друга страна знаем, че curr е единствения нечифтосан елемент в $A[1 \,..\, n]$ и то $\text{cnt} - f > 0$ на брой пъти. Следователно елемента curr ни е мажорантата, което връщаме като резултат на ред 14.

***b*)**

```
1. MajorityElement(A[1 .. n]) : A ∈ ℕ^n, n ∈ ℕ^+
2.     m ← MajorityElementGuarantee(A[1 .. n])
3.     cnt ← 0
4.     for i ← 1 to n
5.         if A[i] = m then
6.             cnt ← cnt + 1
7.     return cnt ≥ ⌊n/2⌋ + 1
```

Коректността остава за упражнение.

Опътване Разгледайте 2 случая - когато имаме мажоранта и когато нямаме.. единия случай е директен, а за другия допуснете противното.