Зад. 1

Нека $U = \{0, \ldots, N-1\}$. Даден е масив $A[1..n] \in U^n : \forall i, j \in \{1, \ldots, n\}$ е изп. $A[i] = A[j] \leftrightarrow i = j$. Нека $\log(|U|) << n << |U|$. Да се съставят двойка бързи алгоритми - индекс и заявка, които да решават следния проблем:

$$\begin{cases} \text{Вход} : k \in \{0, \ldots, N-1\} \\ \text{Изход} : \text{succ}_A(k) \stackrel{\text{def}}{=} \text{IF } \exists x \in A[1..n] : x \geq k \text{ THEN } \min\{x \in A \mid x \geq k\} \text{ ELSE } -1 \end{cases}$$

Пример:

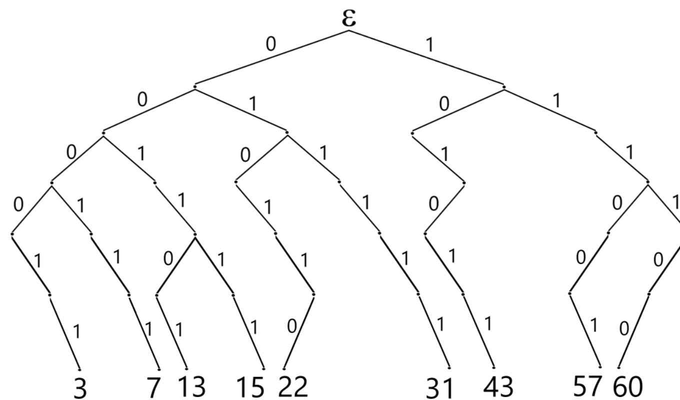
$$\begin{cases} \text{Дадено} : N = 2^{64} - 1,\ n = 9,\ A[1..9] = [3, 7, 13, 15, 22, 31, 43, 57, 60] \\ \begin{cases} \text{Вход} : 1 \\ \text{Изход} : 3 \end{cases} \\ \begin{cases} \text{Вход} : 3 \\ \text{Изход} : 3 \end{cases} \\ \begin{cases} \text{Вход} : 14 \\ \text{Изход} : 15 \end{cases} \\ \begin{cases} \text{Вход} : 62 \\ \text{Изход} : -1 \end{cases} \end{cases}$$

Идея (наивна):

Нека разгледаме идеята за $U = \{0, \ldots, 63\}$ за по-добра илюстрация.

Индекс:

1. Строим си *trie* по числата от масива:

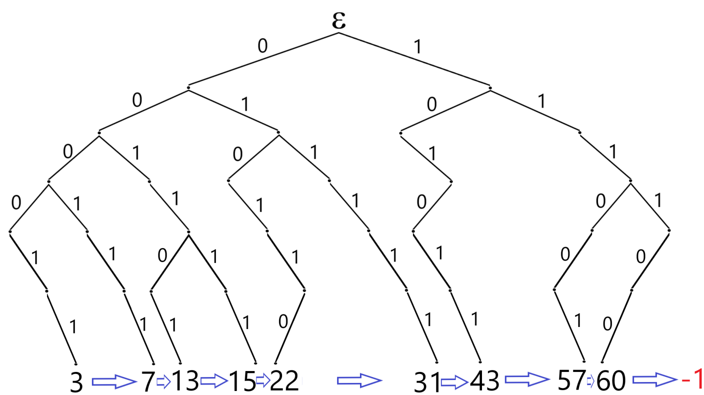


Ясно е, че можем да го построим за $\boldsymbol{O(n.\log(N))}$ време и памет.

2. Към всяко състояние ще добавим указател към най-лявото и най-дясното листо.

Ясно е, че можем да го направим за $\boldsymbol{O(n.\log(N))}$ време и памет.

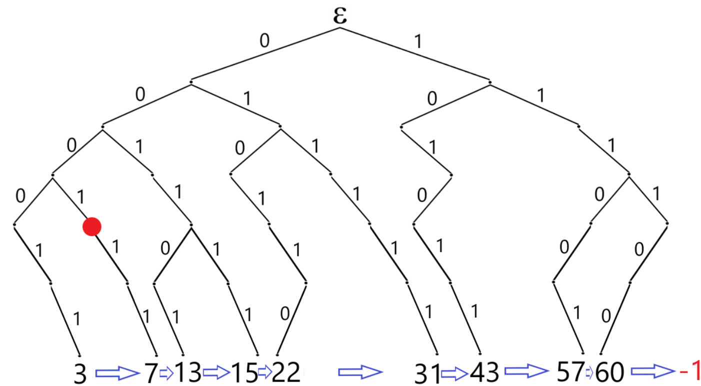
3. Към всяко листо добавяме указател към следващото листо (next (3) = 7, next(7) = 13, …, next(60) = −1):



Ясно е, че можем да го направим за $\boldsymbol{O(n)}$ време и памет.

Заявка:

Нека сме дали заявка за $k = 4$. Прехвърляме $k = 4_{(10)}$ в двоична бройна система и допълваме с нули отпред. Тоест $k = 000010_{(2)}$. Сега намираме върха, който е най-дълъг префикс на $k = 000010_{(2)}$. На картинката отдолу е маркиран с червен цвят.



Да забележим, че върха до който сме достигнали гарантирано е или листо или има точно един наследник. Убедете се.

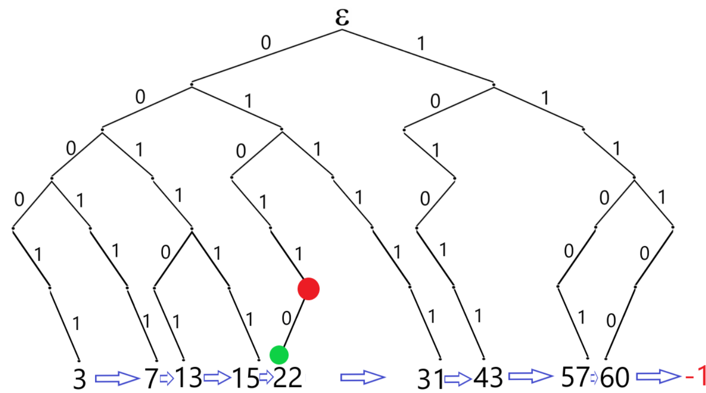
a) Ако е листо сме готови - намерили сме първият по-голям или равен елемент от k в масива $A[1 \,..\, n]$.

b) Ако наследника е с етикет "1", то връщаме *най-лявото листо на наследника*.

Конкретно за примера ще имаме “върни ми най-лявото листо на върха с път от корена $0 - 0 - 0 - 1 - 1$”, т.е. 7.

c) Ако наследника е с етикет "0", то връщаме next(*най − дясното листо на наследника*),.

Нека разгледаме друг пример за илюстрация на ***c***). Нека $k = 23_{(10)} = 010111_{(2)}$. Откриваме върха, който е най-дълъг префикс на $k = 010111_{(2)}$. На картиката отдолу е маркиран с червен цвят.



Връщаме next(*най − дясното листо на поддървото с корен отбелязания със зелено връх*) = next(22) = 31.

Note Може да забележим, че няма значение дали търсим най-дясното листо на поддървото с корен червения връх или най-дясното листо на поддървото с корен зеления връх.. аналогично нямаше значение дали търсим най-лявото листо на поддървото с корен червения или с корен наследника на червения връх в предния пример.

Ясно е, че заявката е със сложност $\boldsymbol{\theta(\log(N))}$.

Това обаче не е голямо подобрение спрямо тривиалното решение със сортировка.. Даже напротив - по-лошо е, защото $\log(n) \le \log(N)$.

Сега ще разгледаме подобрена версия на горната идея.

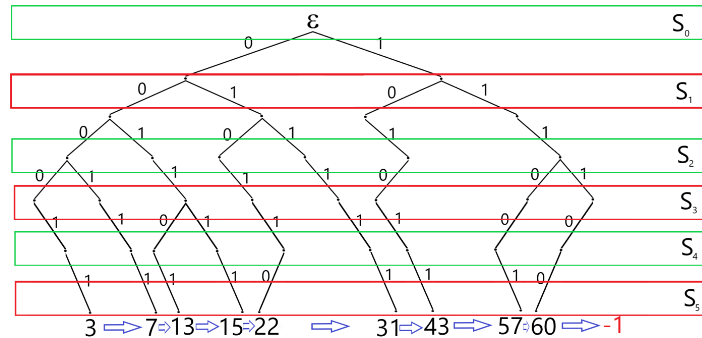
<u>Идея (подобрена):</u>

Нека разгледаме идеята за $U = \{0, \ldots, 63\}$ за по-добра илюстрация.

Идята ни е да правим двоично търсене по височината на дървото. Тогава сложността за зявка би спаднала до $\boldsymbol{O(\log(\log(N)))}$, което ще бъде подобрение на решението със сортировка със сложност $\boldsymbol{O(\log(n))}$.

За целта ще ни трябва по един хеш за всяко ниво на дървото. Тъй като стандартните хешове са с амортизирана сложност $\theta(1)$, а перфектния хеш е със стандартна сложност $\theta(1)$, то ще използваме перфектен хеш.

1. Строим перфектен хеш за всяко ниво на дървото:



Заб Цветовете са различни (зелен и червен) единствено за четимост.

Заб Няма да показваме реализация на перфектния хеш, само ще се възползваме от хубавите му свойста.

Факт Перфектен хеш с размер m се индексира за $O(m)$ очаквано време и заема $\theta(m)$ памет.

Тогава общото време за индексация е $O(\,|\,S_0\,|+|\,S_1\,|+\ldots+|\,S_{\log(N)}\,|) \overset{|S_i|\leq n}{=} \boldsymbol{O(n.\log(N))}$. Като заемаме аналогично $\boldsymbol{O(n.\log(N))}$ памет.

2. Индексираме масив pow$[1\,..\,\lceil\log(N)\rceil]$: pow$[m] = 2^m$.

Ясно е, че това може да го направим за $\boldsymbol{\theta(\log(N))}$ време и също толкова памет.

Заявка:

Нека $k = 23_{(10)} = 010111_{(2)}$. За да вземем префикс с дължина m на числото $k = 010111_{(2)}$ трябва да разделим на 2^{N-m} и да закръглим надолу. Да направим няколко примера:

Искаме префикс с дължина 2. Заместваме във ф-лата и получаваме $\left\lfloor \frac{23}{2^{6-2}} \right\rfloor = \left\lfloor \frac{23}{16} \right\rfloor = 1_{(10)} = 01_{(2)}$. Вярно е, че 01 е префикс на 010111.

Искаме префикс с дължина 4. Заместваме във ф-лата и получаваме $\left\lfloor \frac{23}{2^{6-4}} \right\rfloor = \left\lfloor \frac{23}{4} \right\rfloor = 5_{(10)} = 0101_{(2)}$. Вярно е, че 0101 е префикс.

След като вече сме взели префикса, проверяваме дали се среща в хеша S_m. Ако се среща, то ще търсим в долната половина, ако ли не - в горната.. стандарно двоично търсене.

<u>Псевдокод (някой ден може да го довърша.. очевидно е далече от завършен):</u>

```
Struct Node:
        left ← NULL
        right ← NULL
        leftMost ← NULL
        rightMost ← NULL
        nextVal ← −1
```

```
1.initPow(N) : // N ∈ ℕ
2.      m ← ⌈log(N)⌉
2.      pow[1 .. m] ← Alloc(m)
3.      pow[1] = 1
4.      for i ← 2 to m
5.              pow[i] = 2 * pow[i − 1]
6.      return pow[1 .. m]
```

```
1.getBinary(a) : // a ∈ U
2.      bin[1 .. ⌈log(a)⌉] ← Alloc(⌈log(a)⌉)
```

```
3.	for i ← ⌈log(a)⌉ down to 1
4.		bin[i] ← a %2
5.		a ← ⌊a/2⌋
6.	return bin[1 .. ⌈log(a)⌉]
```

```
1.addToTrie(root, a) : // root is Node, a ∈ U
2.	bin[1 .. ⌈log(a)⌉] ← getBinary(a)
3.	node ← root
4.	leaf ← Node.Init()x
3.	for i ← 1 to bin.size − 1
4.		if bin[i] = 0 then
5.			if node.left = NULL then
6.				node.left ← Node.Init()
7.			node ← node.left
8.		else
9.			if node.right = NULL then
10.				node.right ← Node.Init()
11.			node ← node.right
```

```
1.buildTrie(A[1 .. n]) : // A ∈ U
2.	root ← Node.Init()
```

Амортизирана сложност

Няма да даваме формална дефиниция, но ще дадем интуиция. Не се изисква в настоящия курс, но е важно да бъде чувано от всеки програмист.

За разлика от сложността, която сме дефинирали в текущия курс, амортизираната сложност се среща само при алгоритми от тип "заявка". Идеята им е да дадат средна (в смисъл на математическо очакване) сложност на заявката. Някои заявки биха били много бавни, но за сметка на това рядко срещащи се. Други биха били много бързи и много често срещащи се. Ще разгледаме пример с std::vector.push_back ().

Нека вектора ни е инициализиран с [1, 2, 3, 4].

Нека добавим елемент 5. Тогава получаваме [1, 2, 3, 4, 5, _, _, _], което отнема 9 единици време.

Нека добавим елемент 6. Тогава получаваме [1, 2, 3, 4, 5, 6, _, _], което отнема 1 единица време.

Нека добавим елемент 7. Тогава получаваме [1, 2, 3, 4, 5, 6, 7, _], което отнема 1 единица време.

Нека добавим елемент 8. Тогава получаваме [1, 2, 3, 4, 5, 6, 7, 8], което отнема 1 единица време.

Нека да видим средното време: $\frac{9+1+1+1}{4} = 3$.

Нека добавим елемент 9. Тогава получаваме [1, 2, 3, 4, 5, 6, 7, 8, 9, _, _, _, _, _, _, _], което отнема 17 единици време.

Нека добавим елемент 10. Тогава получаваме [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, _, _, _, _, _, _], което отнема 1 единица време.

…

Нека добавим елемент 16 Тогава получаваме [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16], което отнема 1 единица време.

Нека да видим средното време: $\frac{9+1+1+1+17+1+1+1+1+1+1+1}{12} = \frac{36}{12} = 3$.

Може да забележим, че при $n \to \infty$ средната работа на заявката push_back (k) е 3 единици, т.е константа. Разбира се има заявки, които са линейни, но са с честота $\frac{1}{n}$ и се унищожават. Затова не можем да кажем, че заявките са със сложност $\theta(1)$.. те са $O(n)$.

Идеята на амортизираната сложност е, че ще викаме много на брой заявки и не ни интересува най-лошия случай, а средния. Чрез тях изследването на по-сложни алгоритми става по-лесно. Например:

```
for i ← 1 to n
	vec.push_back (i)
```

Няма да кажем $n.O(n)$ и/или да се чудим по-конкретно, ами директно знаем амортизирано $n * \theta(1)$. Може да забележим, че тук

викаме заявката многократно.