```
BinarySearch (A[1...n]: array of numbers, x: number):
1)  i ← 1
2)  j ← n
3)  while i ≤ j do
4)      k ← ⌊(i+j)/2⌋
5)      if x = A[k]
6)          return k
7)      else if x < A[k]
8)          j ← k-1
9)      else
10)         i ← k+1
11) return -1
```

в тях свързваме "=" numbers

инварианта

Като precondition предполагаме, че масива е A[1..n] е сортиран, защото binarysearch няе работи коректно (предстой за го показнем), ако входния масив е сортиран спрямо релацията "≤" (казвом го така, защото в общия случай може

да имате произволен тип данни, но стига спрямо релацията примерно R, да са линейно наредени (т.е. в редичка да ги подредим кой е I^{ви}, II^{ри} итн.) и операциите „=" и „R", да са ефективни (т.е. да са изчислими с компютър), то ние може в/у тях да извършим binary search при условие, че сме ги сортирали предитова като му ги подаваме.

Сега за инвариантата. Вчера изведохме и ни трябват още две условия (един ваш колега загатна да го кажа едното, но не го взех под внимание).

Инвариантата: На всяко достигане на ред 3) и по точно проверката „$i<j$" е в сила:

1) $j - i \le \frac{n}{2^s}$, s - # влизания досега в цикъла

Т.е. 1) както разстоянието м/у i и j се скъсява всеки път двойно (помага за сметане на сложност)

2) $x \in A[i \dots j]$

Реално това ни е предположението още при стартирането на алгоритъма.

Сега да видим тези условия дали са ни достатъчни:

Индукция по $\underline{s}$:

<u>База</u>: За $s = 0$ (нито веднъж не е изпълнен while цикъла), тогава $i = 1$, $j = n$ и

1) $j - i = n - 1 \leq n = \frac{n}{2^0}$

2) $x \in A[1 \dots n]$ да по предположение при стартиране на алгоритъма

(ИП) Допускаме, че за някое валидно s, което не е последно, то инвариантата е вярна

(Стъпка): Нека сме през текущата итерация на while и съответно сме присвоили ст-тите на променливите, от които зависи инвариантата и отново сме на ред в преди проверката „$i < j$".

Трябва да видим, че е вярна инвариантата т.е.:

1) $j_{new} - i_{new} \leq \frac{n}{2^{s_{new}}}$ $\quad s_{new} = s + 1$

2) $x \in A[i_{new} \dots j_{new}]$

Проверяваме ред по ред какво е могло да се промени:

4) $k \in \left\{\left\lfloor \frac{i+j}{2} \right\rfloor\right\}$

(Сл. 1) $x = A[k]$

$A[k]$ = „$A\left[\left\lfloor \frac{i+j}{2} \right\rfloor\right]$"

Тогава $x \in \{A[i], A[i+1], \dots, A\left[\left\lfloor \frac{i+j}{2} \right\rfloor\right], \dots, A[j]\}$

и алгоритъма приключваме с return k

(сл.2) $x < A[k]$

Тогава $j_{new} = k-1$, $i_{new} = i$

$$j_{new} - i_{new} = \left\lfloor \frac{i+j}{2} \right\rfloor - 1 - i \leq \frac{i+j}{2} - i =$$

$$= \frac{j-i}{2} \overset{(4.n)}{\leq} \frac{n}{2.2^s} = \frac{n}{2^{s+1}} = \frac{n}{2^{s_{new}}}$$

Т.к. $x < A[k]$, то знаем (използваме precondition $A[1..n]$ е сортиран) $x \notin A[k, \dots, j]$ и знаем $+(4.n)$

$x \in A[i, \dots, (k-1)]$, а $j_{new} = k-1$ т.е. $x \in A[i_{new}, \dots, j_{new}]$

(сл.3) $A[k] < x$

Тогава $i_{new} = k+1$, $j_{new} = j$

$$j_{new} - i_{new} = j - \left\lfloor \frac{i+j}{2} \right\rfloor - 1 \leq j - \frac{i+j}{2} =$$

$$= \frac{j-i}{2} \overset{(4.n)}{\leq} \frac{n}{2.2^s} = \frac{n}{2^{s+1}} = \frac{n}{2^{s_{new}}}$$

т.к. $x > A[k]$ (използване precondition $A[1..n]$ е сортиран)

то $x \notin A[i...k]$ и $+(U.\Pi)$, което
деше $x \in A[i...j]$, то
$x \in [(k+1)...j]$, но $i_{new} = k+1$,
$j_{new} = j$
и имаме $x \in [i_{new}...j_{new}]$

Добре, сега се виждаме дали терминира този алгоритъм и дали терминира, той връща ли коректен резултат.

Т.н. $\{j_t - i_t\}_{t \in \mathbb{N}}$ е монотонно (строго) спрямо това на кой итерация сме намаляваща редица, т.е.
$0 \leq j_t - i_t \leq \frac{n}{2^t}$, то за някое
условието в while цикъла $t \in \mathbb{N} \to j_t - i_t < 0$ т.е. $i_t > j_t$ и
$\exists m \in \mathbb{N}: i_t = j_t + m$

Т.е. while цикълът не е безкраен. Всеки път съответно разстоянието м/у тях докато се разминат.

Как да дадем някаква оценка на това колко пъти се е изпълнил.

Ами имаме, че на всяка стъпка, която не е последна е в сила следното: $1 \le j-i \le \frac{n}{2^{s-1}}$

$s-1$ е, защото при $i=j$, то $j-i=0$ и не важи неравенството! (това е специален случай $i=j$ - рядко ще се случва, но трябва да го отчетем)

Тогава $1 \le \frac{n}{2^{s-1}} \Leftrightarrow$

$n \ge 2^{s-1}$ // lg

$\lg n \ge s-1$

т.е. $\lg n - 1 \ge \underline{s}$ — двойвлизания в while цикъла поне $\lg n - 1$ пъти сме влезли

Знаем $O(\log n)$.

Дори и коректност, че времето
правилен резултат?

Вече видяхме терминира,
тъй като не разглеждаме, когато
терминира, как се създаде
алгоритъма.

Нашите очаквания са
следните:

При изпълнение сме предположили, че
хипотеза $x \in A[1..n]$. Искаме алгоритъм
следното да направи:

Ако хипотезата е вярна, то
да върне $k \in \{1, \dots, n\}$ - индекс
на елемента x в масива $A[1..n]$
(като precondition е сортиран).

Ако хипотезата не е вярна, то
да върне -1. Защо -1? Защото
индексите на масива са от 1 до
положителни естествени числа, а
-1 е отрицателно число.

Случаи, в които завършва цикъла.

(Сл.1) За някоя итерация t на while цикъла имаме, че $x \in A[k_t]$, където $k_t = \left\lfloor \frac{i_t + j_t}{2} \right\rfloor$

$O(\log n)$ $x \in A[i_t \dots j_t]$

Тогава правим return k_t и т.к. k_t реално се извършва сред i_t и j_t, то имаме $i_t \le k_t \le j_t$. Сега виждаме, че вече инвариантата ни изисква още само условие и то е, че $1 \le i_t$ и $j_t \le n$. Лесно се доказва защо i_t го увеличаваме само (не строго, на някои итерации не го променяме), а j_t го намаляваме (отново не строго), т.е. никога не намаляме i_t и не увеличаваме j_t, за да спрем. Тогава имаме $1 \le i_t \le k_t \le j_t \le n$ т.е. $1 \le k_t \le n$, което искахме и за накрая.

Другия случай, когато се е спрял

условието за while т.е.
за някое $t \in \mathbb{N}$, то $j_t - i_t < 0$ т.е
$j_t < i_t$. Тогава от инвариантата
имаме, че
$x \in A[i_t \dots j_t] = A[]$
празен масив

т.е. x не е вътре в масива
$A[1 \dots n]$ и не достигнат
до правият return -1, който
ни показва коректно, че
$x \notin A[1 \dots n]$.

Не съм сигурен дали е ясно, че
е толкова добро доказателство
и с вратичка, така че се
извинявам. Ако имате въпроси,
пишете ми.

Смятане на сложности на фрагменти:

Пример 1

```
a ← 0                          const
for i ← 1 to n do
    for j ← 1 to i do
        a ← a + 1              const
return a                       const
```

$1 + 2 + 3 + \ldots + n = n + (n-1) + (n-2) + \ldots + 2 + 1 =$

$= (n+1) \cdot \frac{n}{2} \simeq n^2$

Стойности на цикъла

Пример 2

```
a ← 0
for i ← 1 to n do
    for j ← 1 to n do
        for k ← n+i+j-3 to n do
            a ← a + 1          const
return a                       const
```

За $i=1, j=1$: $k = n-1$ — влиза

За $i=1, j=2$: $k = n$ — влиза

За $i=2, j=1$: $k = n$ — влиза

Трябва да знаем влизания в този цикъл, т.е. сложността му е const.

Така цялостната сложност $\Theta(n^2)$

пример 3) Имаме масив $A = [1, 2, \ldots, n]$, стек S,

предикат $P(S)$ – сложност $\Theta(1)$

↳ true, ако в S има поне 2 елем.

↳ false, иначе

```
push(A[1], S)
push(A[2], S)
for i <- 3 to n do
    while P(S) do   <- инвариантата: всеки път на достигане на проверката за влизане в цикъла в S има поне 2 елемента
        pop(S)
    push(A[i], S)
```

Сложност $T(n) = \Theta(n)$

На страница 13 до 15 на файла, който съм качил допълнително е следващата задача, която изровихме с древния алгоритъм за умножение на две числа наследство от египтяни и етиопци и смятам, че и от детайли към история, и обяснението е по изчерпателно от това, което аз бих описал.

```
alg(A[1,...,n]: array of integers)
  s ← 0
  i ← 1
  j ← n
  while i < j do
      s ← s + A[i] + A[j]
      i ← i + 1
      j ← j - 1
  if n is odd
      s ← s + A[(n+1)/2]
  return s
```

$0 < j - i$

} събира всичките елем. на масива за n-четно

} условие ← добавя за n odd

средния елемент на масива

Тв! Alg връща сумата на всички числа в масива $A[1..n]$

Инв: При всяко достигане на проверката $i<j$:

$$s = \sum_{k=1}^{(i-1)} A[k] + \sum_{k=(j+1)}^{n} A[k] \quad \text{и} \quad i-1 = n-j$$

Д-во инварианта:

База: $s=0$, $i=1$, $j=n$: $s = 0 = \sum_{k=1}^{0} A[k] + \sum_{k=n+1}^{n} A[k] = 0+0$

и $1-1 = n-n \Leftrightarrow 0=0$

(И.П) Нека е в сила за някое достигане, което не е последно.

Сл. пред 5) $S_{new} = S + A[i] + A[j] \overset{И.П}{=}$

$= \sum_{k=1}^{(i+1)} A[k] + \sum_{k=j+1}^{n} A[k] + A[i] + A[j] =$

$= \sum_{k=1}^{i} A[k] + \sum_{k=j}^{n} A[k]$

$i_{new} = i+1$, $j_{new} = j-1$ и старото тях

$S_{new} = \sum_{k=1}^{i_{new}} A[k] + \sum_{k=j_{new}+1}^{n} A[k]$, а

$i_{new} - 1 = i + 1 - 1 = (i-1) + 1 = (n-j) + 1$

$n - j_{new} = n - (j-1) = (n-j+1) \overset{\text{ин}}{=} j$

Терминация:

За последно достигане важи $i \geq j$

(1 сл) не непременно

Тогава $i = j$.

От ин.в: $i - 1 = n - j$ т.е. $i - 1 = n - i \Leftrightarrow$

$n = 2i + 1$

$S = \sum_{k=1}^{i-1} A[k] + \sum_{k=i+1}^{n} A[k]$ и с if-а стойност $S = \sum_{k=1}^{n} A[k]$

(Сл2) n е четно

Тогава $i = j+1$

От инв: $i-1 = n-j \iff j+1-1 = n-j \iff$

$n = 2j$ и $s = \sum_{k=1}^{j} A[k] + \sum_{k=j}^{n} A[k] = \sum_{k=1}^{n} A[k]$.

Другият път ще направим още задачи за доказателство на коректност на итеративни алгоритми и сметане на сложност. Затова мисля да си е 3 часа, за да си наваксаме от 2-те седмици пропуск. Сега ще слагам фрагменти на два от алгоритмите, чиято коректност искам да докажем другия път и си помислете за инварианти:

Зад 1 Кадане: максимална сума на подмасив

```
kadane(A[1...n]: array of numbers): number
1) localMax ← 0
2) globalMax ← -∞
3) for i ← 1 to n do ← инварианта?
4)     localMax ← max(A[i], A[i] + localMax)
5)     globalMax ← max(localMax, globalMax)
6) return globalMax
```

Зад 1 Selection sort

Докажете коректност и изведете сложността на алгоритъма.

```
sort(A[1...n]: array of numbers): void
1) for i ← 1 to n-1 do ← инв. (1)
2)     min ← i
3)     for j ← i+1 to n do ← инв. (2)
4)         if A[j] < A[min] then
5)             min ← j
6)     swap(A[i], A[j])
```

Още една задача от контролно.
Пресметете колко време и колко асимптотична сложност има?

```
do(A[1...n]:array of numbers): set
1) i ← 1
2) while A[i] ≤ 0 and i ≤ n do
3)         i ← i+1
4) if i = n+1
5)     return EmptySet
6) j ← i
7) temp ← A[i]
8) max ← 0, maxi ← 0, maxj ← 0
9) while j ≤ n
10)     if temp > max
11)         max ← temp
12)         maxi ← i
13)         maxj ← j
14)     j ← j+1
15)     temp ← temp + A[j]
16)     if temp ≤ 0
17)         temp ← 0
18)         i ← j+1
19) return (maxi, maxj)
```