4.
Efektivita algoritmů

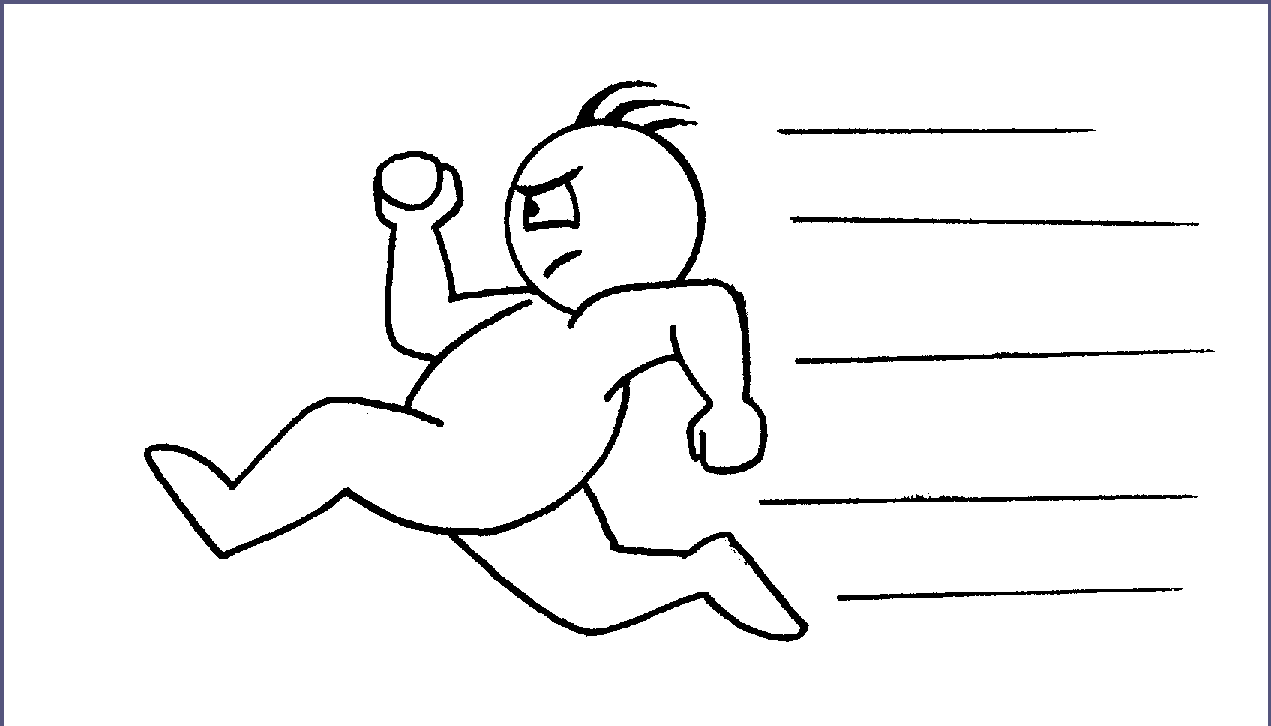
BI-EP1
Efektivní programování 1

ZS 2023/2024

Ing. Martin Kačer, Ph.D.

Katedra teoretické informatiky
Fakulta informačních technologií
České vysoké učení technické v Praze

Jak poznat „rychlý“ program?



(ano, skončí za krátkou dobu)

Proč nestačí stopky…

Doba běhu programu závisí i na jiných okolnostech, než je použitý algoritmus

- **Prostředí** **1**
  - **Jazyk (a další) použitý pro implementaci**
  - **Operační systém, hardware atd.**
- **Vstupní data**
  - **Množství** **2**
  - **Struktura** **3**

Závislost na prostředí

1

- Ignorujeme rozdíly „o konstantu“
- Zjednodušeně řečeno:
  - „na 5x rychlejším HW to poběží 5x rychleji“
- Můžeme si to dovolit?
  1. Nic lepšího nemáme (aspoň ne obecně)
  2. Ostatní rozdíly jsou důležitější (to uvidíme)
  3. Malé konstanty většinou nevadí (2s x 5s)

  - => Obvykle to opravdu můžeme udělat
  - V případě potřeby lze zohlednit i konstanty

„Zanedbávání“ konstant

- Má i další výhody
  - Lze počítat „operace“, nikoli čas
  - Stírá drobné rozdíly

    (někde je pomalejší to, jinde něco jiného)

- Samozřejmě nelze zneužívat
  - 1000x pomalejší bude v praxi vadit

Závislost na množství dat 2

- Nepoužíváme jednu hodnotu, ale funkci
  - Závislost doby běhu na velikosti vstupu
    - 12 n (konstantu ale zanedbáme)
    - n . log n
    - n^2

- => Asymptotická složitost

  (podrobnosti v BI-AG1 a BI-ZDM)

Asymptotická složitost

- Vyjadřuje dobu běhu programu
  - Se zanedbáním konstant pro násobení
  - S přihlédnutím k velikosti vstupu

- Chování při rostoucím objemu dat

(to nás zajímá víc než pro malý vstup)

  - 12ms x 360ms
  - 10s x 40s
  - 1hod x 3hod

Závislost na struktuře dat 3

- Ne všechny vstupy jsou stejně příznivé
  - Nejlepší případ
  - Průměrný případ
  - Nejhorší případ

- Obvykle bereme nejhorší případ
  - Proč?

Složitost pro nejhorší případ

Proč ten nejhorší? Je to totiž:

- **Bezpečné – víme, že program déle nepoběží**
- **Praktické – obvykle se určuje lépe než průměr**
- **Výstižné – často je stejný jako průměr**

- **Průměrný případ – odůvodněné situace**
  - **např. Quick-Sort**

Další měřítka efektivity

- Pomocí asymptotické složitosti lze měřit nároky algoritmu na jakékoli prostředky
  - Cykly procesoru (operace, čas)
  - Spotřeba paměti
  - I/O operace
  - … atd.

Proč operační složitost?

- Operační (časová) složitost nás obvykle zajímá nejvíc
  - Je snad procesor dražší než paměť?
- Ne, ale časová složitost ≥ paměťová
  - (mj. už jen inicializace potřebné paměti)
- Výjimečně i jiná složitost než operační
  - měříme např. I/O, komunikaci atp.

Složitost v číslech

- 1 operace = 1 nanosekunda

složitost	velikost vstupu							
	10	20	50	100	1000	10000	10^6	10^9
$\log_2 n$	3 ns	4 ns	5 ns	6 ns	10 ns	13 ns	20 ns	30 ns
n	10 ns	20 ns	50 ns	0,1 µs	1 µs	10 µs	1 ms	1 s
$n . \log_2 n$	30 ns	80 ns	250 ns	0,6 µs	10 µs	130 µs	20 ms	30 s
$n . \sqrt{n}$	30 ns	80 ns	250 ns	1 ms	30 µs	1 ms	1 s	50 min
n^2	100 ns	0,4 µs	2,5 µs	10 µs	1 ms	100 ms	17 min	32 let
n^3	1 µs	8 µs	125 µs	1 ms	1 s	17 min	32 let	~10^{10} r
n^4	10 µs	160 µs	6 ms	100 ms	17 min	115 dní	~10^7 r	~10^{19} r
2^n	1 µs	1 ms	13 dní	~10^{13} r				
$n!$	4 ms	77 let	~10^{48} r					

Rychlost počítačů

- Má efektivita algoritmů význam v době rychlých počítačů?

- Rozhodně ano!
  1. Při exponenciální složitosti nám žádný HW nepomůže!!
  2. Rychlejší počítače => Více dat
     => Posun v tabulce „doprava”
     => Na efektivitě záleží naopak ještě víc!

Efektivita – praktické dopady

- Vždy nutno brát reálné okolnosti
  - Očekávaný objem dat
  - Asymptotická složitost algoritmu
    - Časová (operační)
    - Paměťová
  - Rozdíly v pracnosti
    - (ano, i to rozhoduje)

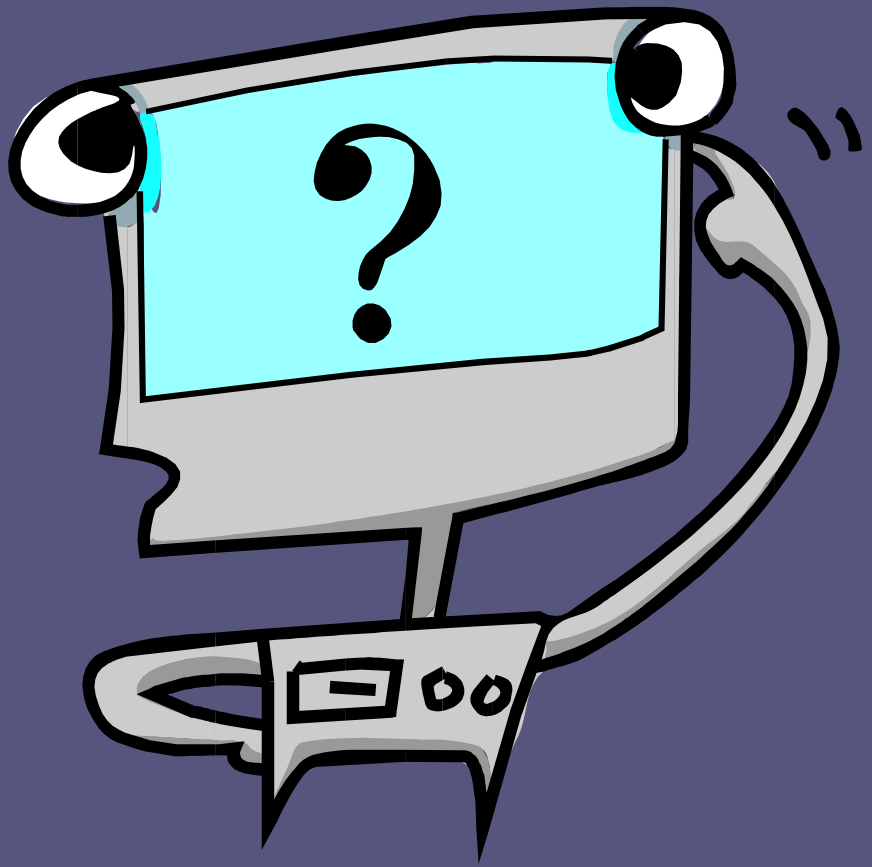
- → Měli bychom znát omezení

Efektivita x velikost vstupu

- Předpokládáme cca miliardy operací
- Hranice samozřejmě není ostrá

Složitost	Omezení na data
$O(n)$	~ 1 000 000 000
$O(n \cdot \log n)$	~ 50 000 000
$O(n \cdot \sqrt{n})$	~ 1 000 000
$O(n^2)$	~ 50 000
$O(n^3)$	~ 2 000
$O(n^4)$	~ 500
$O(2^n)$	~ 30
$O(n!)$	~ 10

Dotazy?



Příklad: hledání řetězce

Vyhledávání – přímá cesta

- Zkouším postupně jednotlivé pozice

```
i + vzorek.length <= text.length
```

```
char[] text, vzorek;
for (int i = 0; i < text.length; ++i) {
      if (test(text, vzorek, i))
            return i;
}
return -1;

boolean test(char[] text, char[] vz, int i) {
      for (int j = 0; j < vz.length; ++j)
            if (text[i+j] != vz[j]) return false;
      return true;
}
```

Vyhledávání – složitost

- Nejhorší případ
  - O(n.m)

- Může nastat?
  - Bohužel opravdu ano
  - Text: aaaaaaaaaaaaaaaaaaaaaaaab
  - Vzorek: aaaaaaaaaab

Vyhledávání – jak na to?

- Jak tedy efektivně vyhledávat?
  - Automaty
  - Efektivní vyhledávací algoritmy

- Viz jiné předměty …
- … a možná také BI-EP2

Určení operační složitosti

Určování složitosti

- **Operace – pozor, zda jsou konstantní**
- **Cyklus – počet opakování**
  - **Někdy nemusí být snadné určit**
- **Rekurze – složitější (viz BI-ZDM)**
- **Vnořené cykly – násobí se**
  - **Co když se počet opakování vnitřního cyklu mění?**

Určování složitosti

- Maximální počet opakování
  - Důležitý je celkový možný počet operací, nemusí odpovídat vnoření cyklů

- Příklad – dělení seřazené posloupnosti
  - Vstup: 1 4 7 10 17 18 20 24
    (rozdělit podle: 5 10 16 21)
    1 4 / 7 10 / / 17 18 20 / 24

```
a[] = {1, 4, 7, 10,
       17, 18, 20, 24};

b[] = {5, 10, 16, 21};
```

```
1 4
7 10

17 18 20
24
```

Určování složitosti – příklad

- Posouvám index v poli a testuji

```
int a[MAX], b[MAX], an, bn, ia, ib;

ia = 0;
for (int ib = 0; ib < bn; ++i) {
      while (ia < an && a[ia] <= b[ib])
            printf("%d ", a[ia++]);
      printf("\n");
}
while (ia < an)
      printf("%d ", a[ia++]);
printf("\n");
```

O(an * bn)

Určování složitosti – příklad

- Vnořené cykly, ale složitost lineární

```
int a[MAX], b[MAX], an, bn, ia, ib;

ia = 0;
for (int ib = 0; ib < bn; ++i) {
      while (ia < an && a[ia] <= b[ib])
            printf("%d ", a[ia++]);
      printf("\n");
}
while (ia < an)
      printf("%d ", a[ia++]);
printf("\n");
```

Maximální počet opakování: an (celkem!)

O(an + bn)

Určování složitosti – příklad 2

- Nejkratší souvislý úsek v posloupnosti kladných čísel se součtem ≥ MIN

MIN=16

Určování složitosti – příklad 2

- Ukazatel na začátek a konec úseku
  - Součet malý => přidávám další na konec
  - Součet vyhovuje => odebírám ze začátku

```
int a[LEN], start = 0, end = 0, sum = 0, best = 0;

while (end <= LEN) {
      while (sum < MIN && end < LEN) {
            sum += a[end++];
      }
      if (end - start > best && sum >= MIN)
            best = end - start;
      while (sum >= MIN) {
            sum -= a[start++];
}     }
```

O(LEN)

Maximální počet opakování: LEN (celkem!!)

Určování složitosti – příklad 2

- Pro porovnání – jediný cyklus
  - Ale dvě „řídící proměnné“,
    tj. i tady není určení složitosti úplně triviální

```
int a[LEN], start = 0, end = 0, sum = 0, best = 0;

while (end <= LEN) {
      if (sum < MIN) sum += a[end++];
      else {
            if (end - start > best)
                  best = end - start;
            sum -= a[start++];
}     }
```

Opakující se operace

Efektivita častých operací

- Porovnáváme následující případy:
  - O(n)
  - O(log n)

- U jediného výpočtu zase tolik nevadí
- Operace se opakuje – významný rozdíl!
  - Volání v cyklu (n volání: n^2 x n.log n)
  - … ale třeba i vytížená webová aplikace!

Časté operace chceme O(log n)

- Snaha o efektivní datové struktury
  - Stromy
  - Haldy
  - Rozptylovací tabulky (hash)

- Předmět BI-AG1
  - … a také příště zde – příklady

Maximální složitost operací

- Příklad – datová struktura „seznam“
  - Alokované pole + počet prvků v něm
  - Operace „přidej na konec“
    - Pokud pole nestačí, musím ho zvětšit

```
int size = 10, values[] = new int[size], count = 0;

void add(int x) {
        if (count < size)
                values[count++] = x;
        else
                . . .
}
```

Seznam – zvětšování pole

- Zvětšování pole – o kolik?
  - Vždy o 1
  - Vždy o 10
  - Vždy dvojnásobek
  - ... a další možnosti

```
size += ???;
int[] nval = new int[size];
for (int i = 0; i < count; ++i) {
      nval[i] = values[i];
}
values = nval;
```

Seznam – vliv na složitost

- Zvětšování o 1
  - Průměrná složitost = maximální = O(n)
- Zvětšování o 10
  - Maximální složitost = O(n)
  - Průměrná složitost = O(n/10) = O(n)
- Zvětšování na dvojnásobek
  - Maximální složitost = O(2*n) = O(n)
  - Průměrná složitost = ?????

Seznam – průměrná složitost

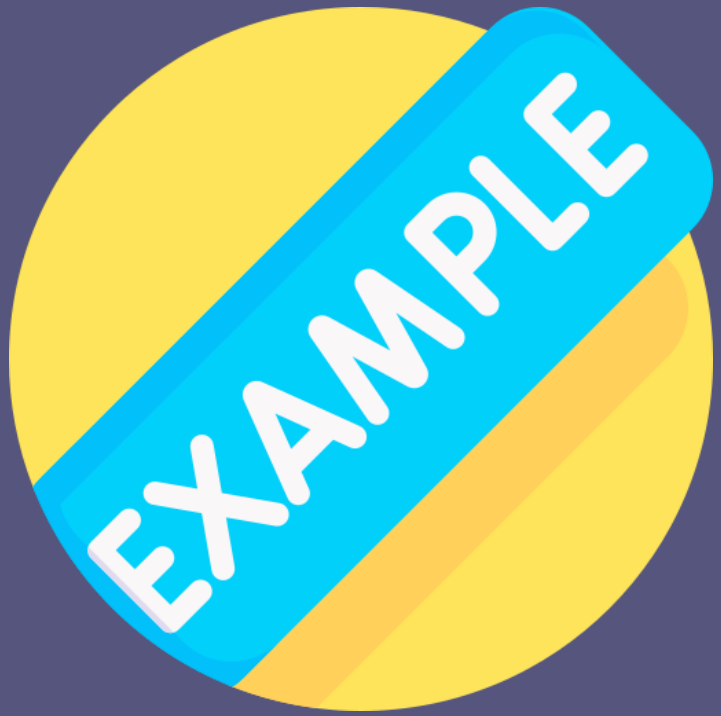
zvětšení	přidání	kopírování	Σ celkem	**průměr**
10 → 20	11	10	10	**0,90**
20 → 40	21	20	30	**1,42**
40 → 80	41	40	70	**1,70**
80 → 160	81	80	150	**1,85**
160 → 320	161	160	310	**1,92**
320 → 640	321	320	630	**1,96**
640 → 1280	641	640	1270	**1,98**
$10.2^n \rightarrow 20.2^n$	$10.2^n + 1$	10.2^n	$20.2^n - 10$	**< 2**

- **Průměrná složitost = konstantní = O(1)**

Amortizovaná složitost

- „Průměrná“ při opakování operací
  - Zaručená!
- Jakákoli posloupnost m operací vede maximálně na složitost m * O(1)
- => „Amortizovaná“ složitost jedné operace je O(1)
  - Kdy rozdíl hraje roli? (amortizovaná x skutečná)
    - Real-time systémy, …

Ještě jeden příklad



Význam příkladu

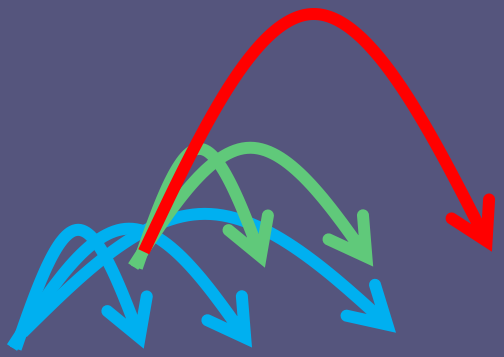
- Přemýšlejte nad datovými strukturami
  - I jednoduché nám poslouží (pole)
  - Uspořádat tak, aby operace byly rychlé
  - Nebojte se *redundantních* dat

Příklad I – Duplicity v poli

- Je zadáno pole čísel
- Připravte si datovou strukturu
- Určete, zda se mezi dvěma indexy nachází některé číslo dvakrát

Duplicity – naivní přístup

- Porovnat každý s každým

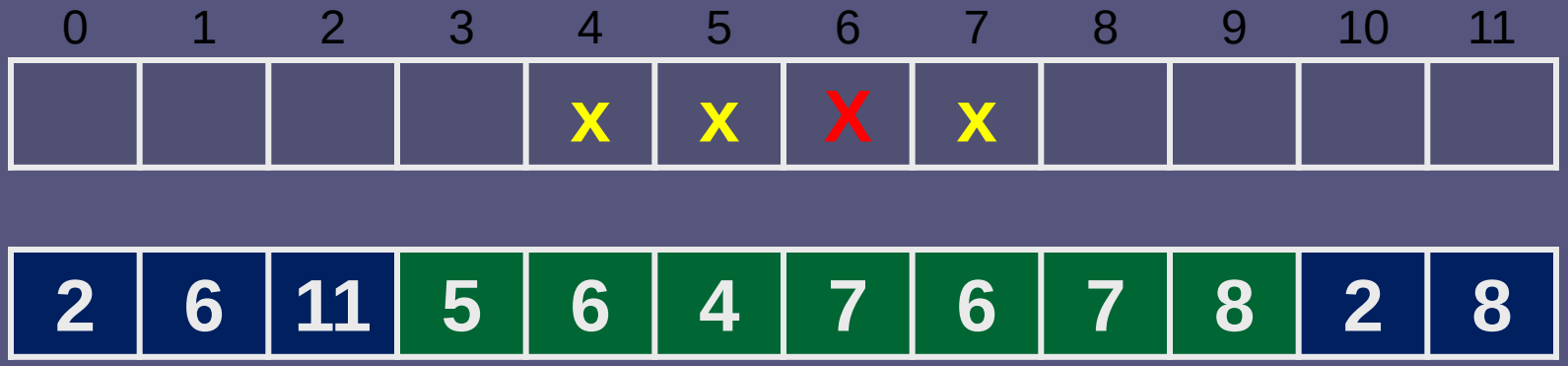


2	6	11	5	6	4	7	6	7	8	2	8

- Jeden dotaz = O(*délka intervalu ^ 2*)
- Jak to udělat lépe?

Duplicity – lepší přístup

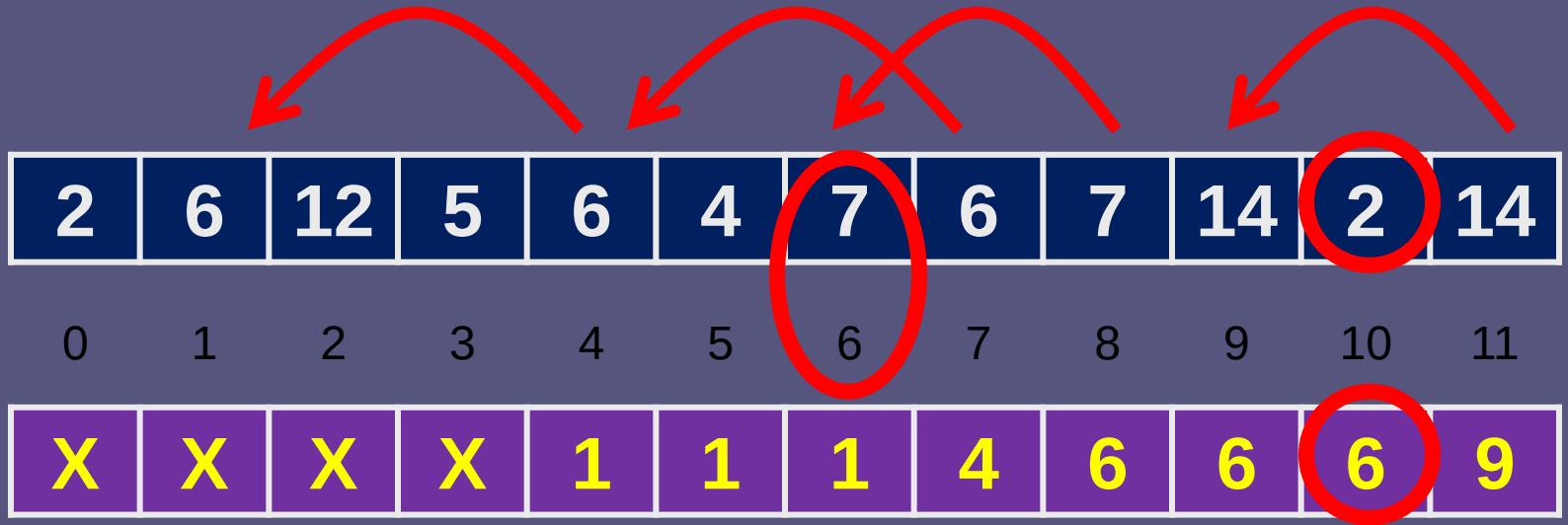
- Projít interval
- Duplicity detekovat v poli (množině)



- Jeden dotaz = O(*délka intervalu*)
- Jak to udělat ještě lépe?

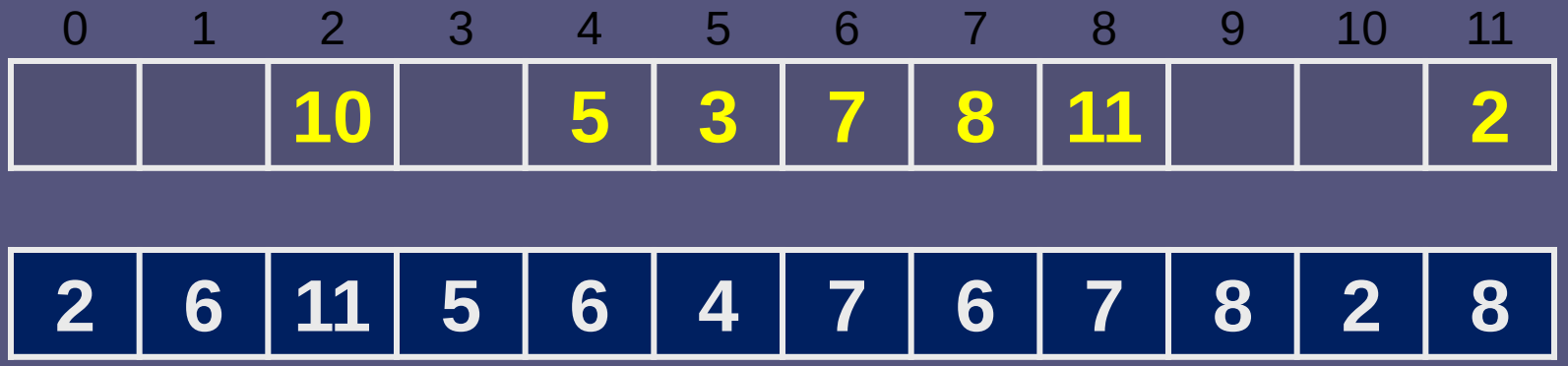
Duplicity – možné řešení

- Pro každý prvek si budeme pamatovat *poslední předchozí duplicitu*



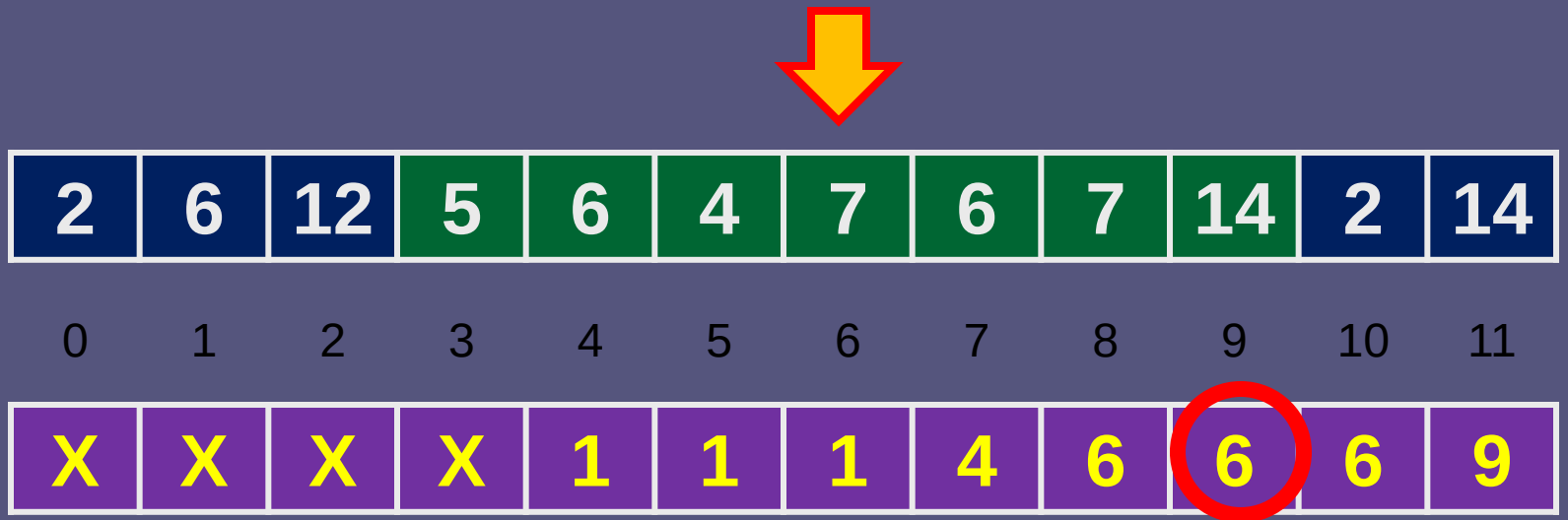
Duplicity – získání dat

- Pro každé číslo si pamatujeme poslední předchozí výskyt
  - Pole nebo tabulka



Duplicity – zjištění řešení

- V konstantním čase
  - Porovnáním začátku s poslední duplicitou



Duplicity – zjištění řešení

- V konstantním čase
  - Porovnáním začátku s poslední duplicitou

Duplicity – kód

- Vytvoření pole

```
for (int i = 0; i < cnt; ++i) {
   idx = previous[nums[i] = nextInt()];
   previous[nums[i]] = i;
   if (idx >= 0 && idx > prvlast)
       prvlast = ii;
   last[i] = prvlast;
}

if (last[b] >= a)
   // duplicita je nums[last[b]]
```

Duplicity – kód

- Ověření duplicit

```
for (int i = 0; i < cnt; ++i) {
   idx = previous[nums[i] = nextInt()];
   previous[nums[i]] = i;
   if (idx >= 0 && idx > prvlast)
        prvlast = ii;
   last[i] = prvlast;
}
a = nextInt(); b = nextInt();
if (last[b] >= a)
   // duplicita je nums[last[b]]
else
   // bez duplicit
```

Duplicity – časová složitost

- Příprava pole
  - Přímé adresování pole výskytů: O(n)
  - S použitím tabulky: O(n . log n)
- Jeden dotaz na duplicity
  - Konstantní: O(1)

Dotazy?