Merging og hashing

Mål

Målet for disse slides er at diskutere nogle metoder til at gemme og hente data effektivt.

Interfaces

Større programmer opbygges ofte i moduler med klare interfaces (grænseflader) til hinanden.

Et interface består af en samling funktioner, som tilbydes til omverdenen (andre dele af samme program, andre programmer). Omverdenen behøver ikke kendskab til implementation af en funktion for at kunne bruge den.

Analogi: interfaces i biler.

Funktion	**Implementation**
Speeder (køre)	Elmotor, benzinmotor, dieselmotor, ...
Bremsepedal (stoppe)	Skivebremse, tromlebremse, ...
Rat (styre)	Hydraulisk, wire, elektrisk, ...

Et interface separerer anvendelse fra implementation (abstraktion). En implementation kan ændres, uden at brugere af interfacet behøver vide det.

Interfaces

Et interface kan være simpelt eller avanceret:

Avancerede interfaces kræver avancerede implementationer.

Simple interfaces kan bruge både simple og avancerede implementationer.

Datastrukturer

Datastruktur = metode til at gemme og hente data effektivt.

Datastrukturer kan beskrives på to niveauer:

Interface: Hvordan kan man tilgå data fra andre dele af et program (hvilke funktioner på data tilbydes)?

Implementation: Hvordan skal disse funktioner implementeres (hvordan skrives kode, som udfører dem)?

Dagens emne: to udbredte interfaces blandt datastrukturer:

- Sekventiel tilgang (simpelt)
- Random access (mere avanceret)

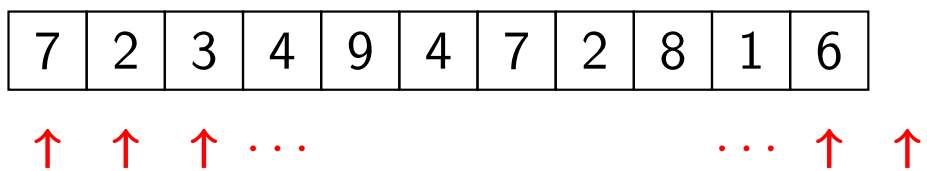
Interface for sekventiel tilgang

Funktioner til læsning:

`readNext()`, `isEndOfFile()`, `open()`, `close()`

Typisk anvendelse:

```
file = open(filename)
Sålænge file.isEndOfFile() svarer False:
   x = file.readNext()
   (...gør noget med x...)
file.close()
```



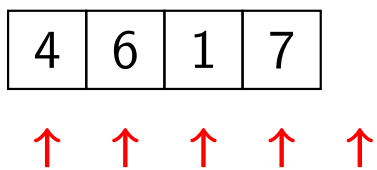
Interface for sekventiel tilgang

Funktioner til skrivning:

`writeNext()`, `open()`, `close()`

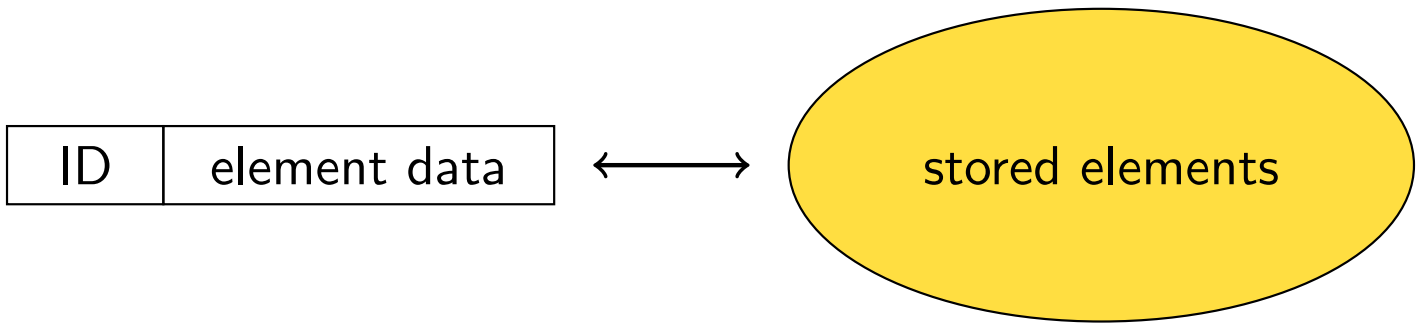
Typisk anvendelse:

```
file = open(filename)
Sålænge der er mere data at generere:
   (...generer næste stykke data x...)
   file.writeNext(x)
file.close()
```



Interface for random access

Tilgang via ID (også kaldet key, nøgle) for elementer.



Funktioner:

`findElm(ID)` (returnerer element data)
`insertElm(ID,data)`
`deleteElm(ID)`
`open()`, `close()`

Interface for random access

```
findElm(ID)
insertElm(ID,data)
deleteElm(ID)
open(), close()
```

Eksempler:

- Databaser (ID = CPR-nummer, f.eks.)
- Dictionaries i Python (gennemgås snart i DM594)
  `data = myDict[ID]`
  `myDict[ID] = data`
  `myDict.pop(ID)` (eller `del myDict[ID]`)
- Lister i Python kan ses som et specialtilfælde, hvor ID = index
  `data = myList[7]`
  `myList[7] = data`
  `myList[7] = None`

Dagens spørgsmål

1. *Hvad kan* det simple interface Sekventiel tilgang *bruges til?*
2. *Hvordan kan man implementere* det mere avancerede interface Random access ?

Bemærk: Alle datakilder kan tilgås med sekventiel tilgang. (Og for nogle kan *kun* sekventiel tilgang laves effektivt.)

- Harddisk
- SSD (Solid State Disk)
- CD
- Bånd
- Streaming over net
- Data genereret on-the-fly af et andet program
- Data i en liste (Python) eller i et array (Java).

Spørgsmål 1: Hvad kan laves med sekventiel tilgang?

F.eks.:

- Lineær søgning (jvf. Lenes slides)
- Find mindste element
- Find sum og antal af elementer (og dermed gennemsnit)
- Fjern mindste element
- Selectionsort (*gennemgås om lidt*)
- Merge: slå to sorterede lister sammen til én sorteret liste (*gennemgås om lidt*)
- Mergesort (*gennemgås om lidt*)
- Generaliseret merge (*gennemgås om lidt*)

Morale: Man kan lave forbavsende mange algoritmer baseret på det simple interface sekventiel tilgang.

Selectionsort

Find mindste element via et scan af input:

7	3	5	8	5	9	2	4	6	4	2	7

I dette tilfælde er mindste element 2. Udtag nu dette fra listen via et scan mere, hvor det første 2 skrives til output, mens alle andre elementer kopieres til en ny liste med de resterende elementer:

2

7	3	5	8	5	9	4	6	4	2	7

Gentag nu ovenstående. Efter seks runder vil vi f.eks. have følgende situation:

2	2	3	4	4	5

7	8	5	9	6	7

Påstand: efter n runder vil input være blevet sorteret:

2	2	3	4	4	5	5	6	7	7	8	9

Selectionsort, korrekthed

Korrekthed af Selectionsort (påstanden) er måske oplagt (vi udtager først det mindste element, så det næstmindste, så det trediemindste,...).

Men vi kan også bevise det ved hjælp af en invariant (cf. Lene's slides):

> i) Elementerne i output er sorteret stigende.
>
> ii) Ingen resterende elementer er mindre end noget element i output.

1) Efter første runde er i) sand [da output har længde én] og ii) er sand [da vi flyttede et mindste element].
2) Antag at i) og ii) er sande *før* en runde. Så er i) sand *efter* runden [i) + ii) + vi stiller det flyttede element sidst i output]. Og ii) er også sand *efter* runden [ii) + vi flyttede et mindste element].

Fra 1) og 2) følger, at i) og ii) er sand efter *alle* runder [induktion]. Også efter sidste runde, hvor alle elementer er i output, som ifølge i) er sorteret.

Selectionsort, køretid

Køretid for Selectionsort? Den er proportional med

$$n + (n-1) + (n-2) + \cdots + 3 + 2 + 1$$

da vi først scanner en liste af længde n, dernæst en liste af længde $n-1$, dernæst en liste af længde $n-2$, etc.

Kald ovenstående sum for S. Så gælder

$$\begin{array}{rccccccccccccc} 2 \cdot S = & n & + & (n-1) & + & (n-2) & + & \ldots & + & 3 & + & 2 & + & 1 \\ + & 1 & + & 2 & + & 3 & + & \ldots & + & (n-2) & + & (n-1) & + & n \end{array}$$

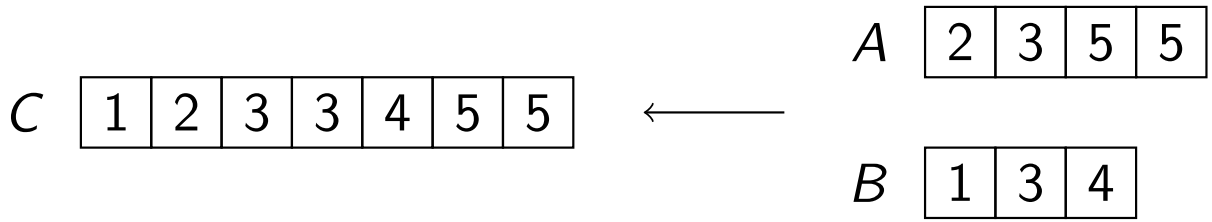
Dvs. $2 \cdot S = n(n+1)$, hvoraf vi ser $S = n(n+1)/2 = n^2/2 + n/2$.

Da n^2 dominerer (vokser hurtigere end) n, er køretiden

$$O(n^2)$$

Merge

Mål: Slå to sorterede lister A og B sammen til én sorteret liste C.



Et merge-skridt: Sammenlign (nuværende) forreste element i A og B, og flyt det mindste af disse over sidst i C.

Start med tom C
Sålænge både A og B er ikke-tomme:
    Udfør et merge-skridt
Hvis enten A eller B er ikke-tom:
    Flyt resten af dens elementer over sidst i C

Er det en korrekt metode (dvs. er C ved afslutning sorteret)?

Ja: Merge er et specialtilfælde af Selectionsort (et merge-skridt finder det mindste blandt de resterende elementer, da inputlisterne er sorterede).

Tid: Er proportional med antal flyttede elementer, dvs. $O(|A| + |B|)$.

Mergesort

Runde 1: Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2 ($= 2^1$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $O(n)$.

Runde 2: Merge de sorterede lister af længde 2 til sorterede lister af længde 4 ($= 2^2$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $O(n)$.

Runde 3...: Længde 4 til længde 8 ($= 2^3$),...

$$\boxed{\text{Efter } i \text{ runder: længde} = 2^i.}$$

Indtil længde n nås, dvs. alle elementer er i én sorteret liste. Dvs. vi har sorteret elementerne.

Der er lavet $\log_2 n$ runder, når længde n nås [da $2^i = n \Leftrightarrow i = \log_2 n$].

Tid i alt: $\boxed{O(n \log_2 n)}$ [da vi $\log_2 n$ gange laver $O(n)$ arbejde].

Mergesort

Illustration af algoritme og analyse:

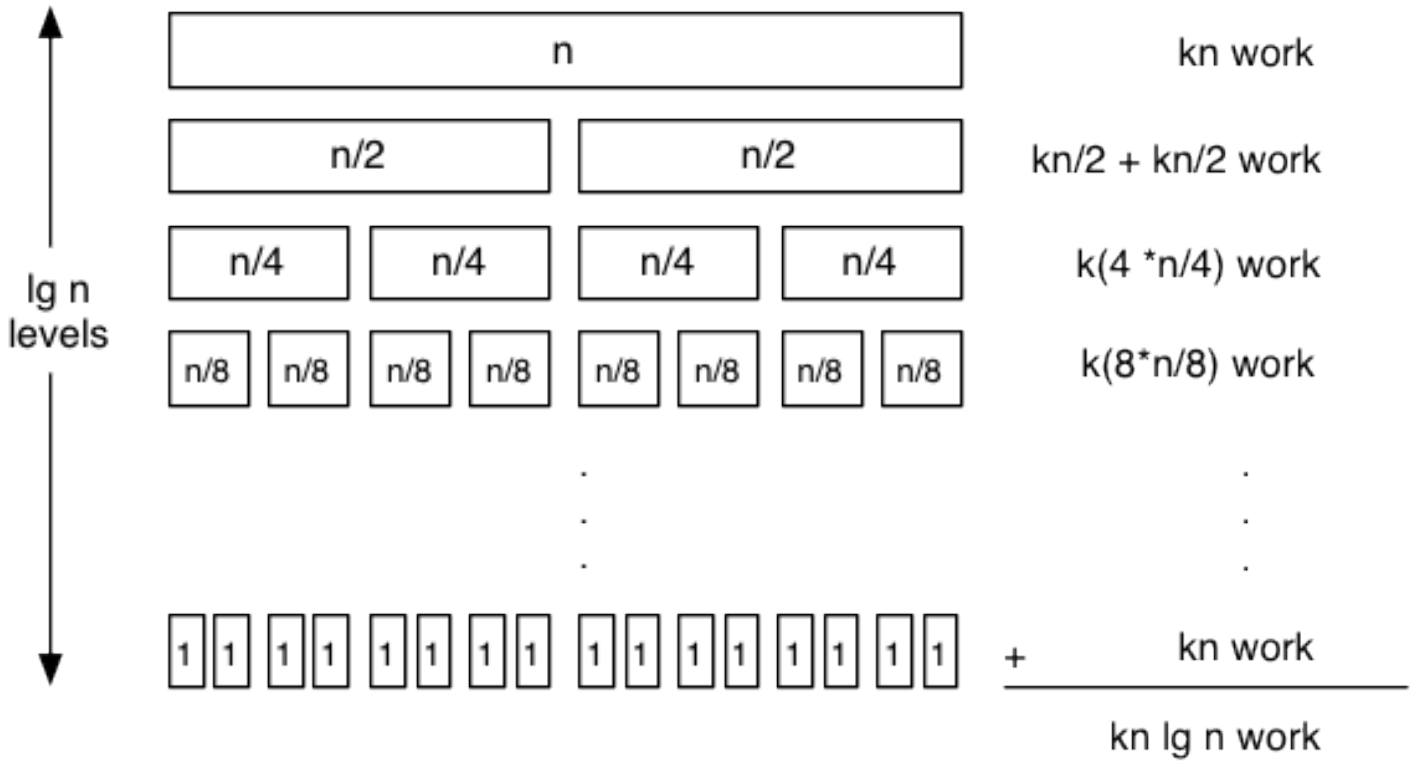


Figure: Andrew Myers, Cornell University

Generaliseret merge

Under merge af to sorterede lister A og B vil elementer med samme værdi "møde" hinanden under merge-skridtene (dvs. de vil på et tidspunkt stå forrest samtidigt i A og B). Dette kan udnyttes:

Eksempel 1: A er datafil med (ID,data)-elementer, sorteret efter ID. B er liste af (ID,dataopdatering)-elementer, sorteret efter ID.

At gennemløbe A og B via merge-skridt vil tillade os at opdatere data i A. NB: de to lister behøver ikke være lige lange.

Eksempel 2: To sorterede lister A og B repræsenterer mængder (og er derfor hver især uden dubletter).

At gennemløbe A og B via merge-skridt vil tillade os at generere f.eks. $A \cap B$: Hvis et merge-step ser to ens elementer som forreste i A og B, flyttes det ene over sidst i C, og det andet smides væk. Hvis et merge-step ser to forskellige, smides det mindste væk.

Spørgsmål 2: Hvordan laves interface Random access?

Én metode: Hashing.

 ?

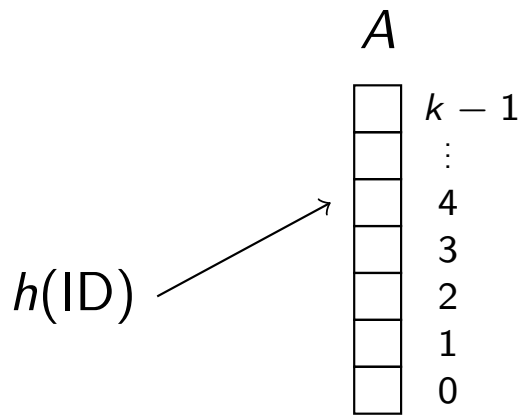
Hash $\sim$ hacher [fransk] $\sim$ hakke i stykker.

Hashing

Ide: Tildel hver ID et index i et array (Python: en liste) A, hvor det tilhørende element gemmes.

Hash-funktion h:

$$h(\text{ID}) = \text{index i } A \text{ for ID}$$



Eksempel (antag ID'er er heltal):

$$h(x) = x \bmod k$$

hvor $k = |A|$ og mod (modulus) betegner rest ved heltalsdivision.

Med $k = 41$:

$$\begin{aligned} h(46) &= 5 && (\text{da } 1 \cdot 41 + 5 = 46) \\ h(12) &= 12 && (\text{da } 0 \cdot 41 + 12 = 12) \\ h(100) &= 18 && (\text{da } 2 \cdot 41 + 18 = 100) \end{aligned}$$

Bemærk at $h(x) \in \{0, 1, 2, \ldots, k-1\}$, så det er altid et lovligt index.

Hashing

Når nu vi antager, at ID'er er heltal, hvorfor så ikke bare bruge

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.

CPR-numre: 180781-2345 $\in \{0, 1, 2, \ldots 10^{10} - 1\}$

Så med ovenstående $h(x)$ skal størrelsen k af A skal være 10^{10} for at gemme 5 tal. Spild af plads (10^{10} bytes = 10 Gb).

Ofte er nøgler heltal (32 eller 64 bits), dvs. har $2^{32} \approx 0.43 \cdot 10^{10}$ eller $2^{64} \approx 0.18 \cdot 10^{20}$ muligheder. Dvs. samme situation eller værre.

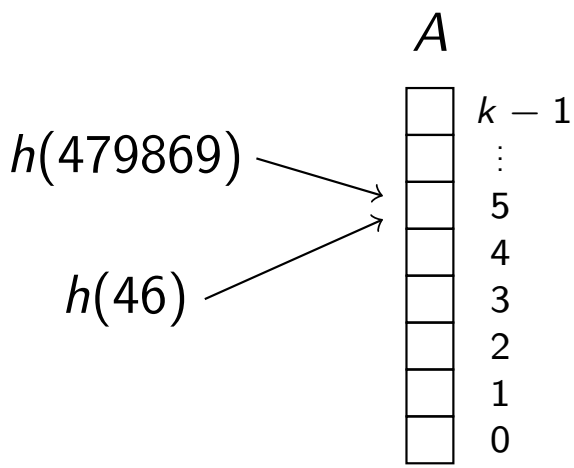
Generelt ønsker vi, at længden k af A cirka er lig antal gemte elemeneter n (så hash-tabellen selv ikke fylder væsentlig mere end de gemte elementer).

Dette opnås ved at vælge k i nærheden af n i udtrykket $h(x) = x \bmod k$. [Om nødvendigt må hash-tabellen genopbygges med større k, hvis n vokser for meget pga. indsættelser.]

Kollisioner

$h(x) = x \bmod 41$:

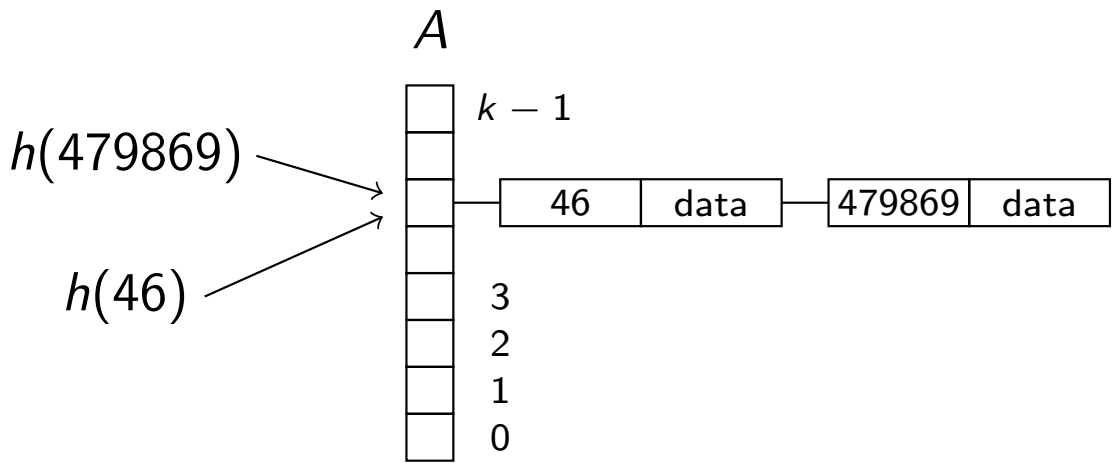
$h(46) = 5$ (da $1 \cdot 41 + 5 = 46$)
$h(12) = 12$ (da $0 \cdot 41 + 12 = 12$)
$h(100) = 18$ (da $2 \cdot 41 + 18 = 100$)
$h(479869) = 5$ (da $11704 \cdot 41 + 5 = 479869$)



Problem: flere elementer vil bo på samme plads i A.

Én løsning: Chaining

Gem alle elementer på samme plads i en liste.



Da listen skal gennemløbes sekventielt (under søgning og under sletning), stiger tiden fra $O(1)$ til $O(|\text{liste}|)$.

Vi vil derfor gerne have få kollisioner.

Kan kollisioner undgås?

Det afhænger af forholdet mellem hash-funktionen h og det konkrete datasæt, som er gemt.

Nogle observationer:

- I værste fald hash'er alle n elementer til samme plads. Tid: $O(n)$.
- Hvis n (antal elementer gemt) er større end k (størrelsen af A), er der mindst én kollision (duehulsprincippet).

Hvad hvis vi antager at h "indsætter tallene på tilfældige pladser"?

NB: Dette er ikke en brugbar hash-funktion, da vi ikke kan finde elementerne igen (vi kan ikke huske, hvor de er lagt). Men det giver en jævn spredning i hash-tabellen for *alle* datasæt, og kan derfor ses som en opførsel, som vi ønsker for en hash-funktion.

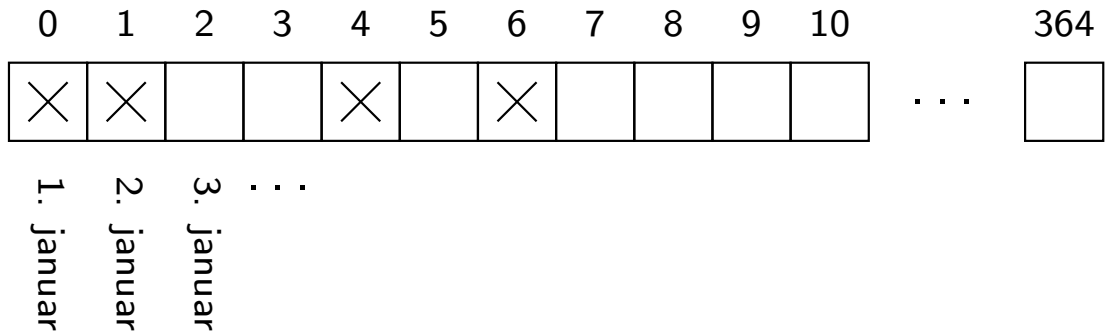
Vi studerer derfor nu denne situation, og ønsker at sige noget om, *hvor hurtigt* kollisioner opstår, når vi indsætter elementer tilfældigt.

Fødselsdage og hashing

Situation: n tilfældige personer går ind i et rum.

Fokus: Er der nogen, som har fødselsdag samme dato i året?

Bemærk: at samle personer med tilfældige fødselsdage svarer til at indsætte tilfældigt i en hash-tabel af størrelse 365 (= antal dage i et år).



Ovenfor er fire personer samlet (hvoraf ingen har fødselsdag samme dag).

Dette er altså et eksempel på den (hashing) situation vi ønsker at studere.

Fødselsdage og hashing

Situation: n tilfældige personer går ind i et rum.

Fokus: Er der nogen, som har fødselsdag samme dato i året?

Sandsynligheden for dette må stige, når n vokser:

n	Sandsynlighed for (mindst) to med samme fødselsdag
0	0
1	0
2	$1/365$
$\vdots$	$\vdots$
$\boxed{?}$	$1/2$
$\vdots$	$\vdots$
366	1

Spørgsmål: For hvilket n bliver sandsynligheden for, at nogen blandt de første n personer har samme fødselsdag, *større* end $1/2$?

Fødselsdage og hashing

Spørgsmål: For hvilket n bliver sandsynligheden for, at nogen blandt de første n personer har samme fødselsdag, større end $1/2$?

Bemærk:

$$\mathrm{P}(\text{nogen har samme fødseldag}) + \mathrm{P}(\text{ingen har samme fødseldag}) = 1$$

Når den første sandsynlighed stiger, må den anden sandsynlighed falde.

Så for at svare på spørgsmålet kigger vi i stedet på følgende sandsynlighed

$$\mathrm{P}(\text{ingen med samme fødseldag blandt de } n \text{ første personer}),$$

og spørger, hvornår den er *mindre* end $1/2$ (da $1/2 + 1/2 = 1$).

Fødselsdage og hashing

Ingen med samme fødseldag blandt de n første personer

$\Updownarrow$

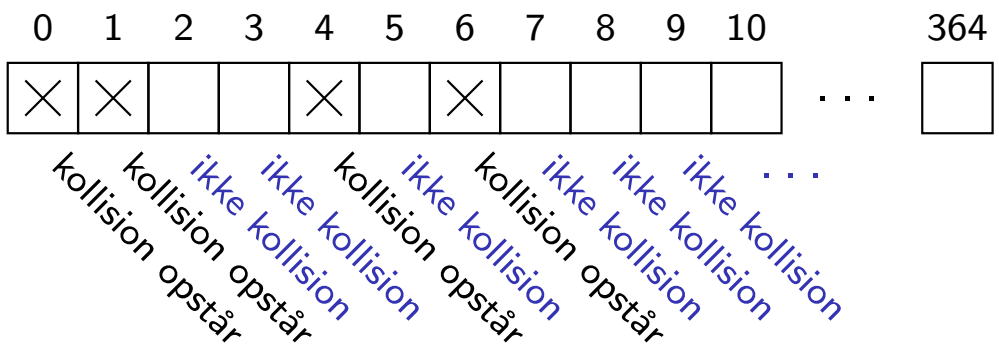
1) Ingen med samme fødselsedag blandt de $n-1$ første personer

og

2) Den n'te persons fødselsdag er ikke lig nogen af disses.

Hvis 1) gælder, er præcis $n-1$ forskellige datoer optaget, når den n'te person går ind i rummet. Hvis personerne antages at have tilfældige fødselsdage, er sandsynligheden for 2) derfor $(365-(n-1))/365$.

Situationen for $n=5$ (dvs. 4 personer allerede til stede, uden kollision):



Fødselsdage og hashing

Hvis vi kalder P(ingen med samme fødseldag blandt de n første personer) for p_n, kan vi se af ovenstående at

$$p_n = p_{n-1} \cdot \frac{365 - (n-1)}{365}$$

Da p_1 naturligvis er 1 (med kun én person i rummet er der helt sikkert ingen med samme fødselsdag), ser vi at:

$$\begin{aligned}
p_1 &= 1 \\
p_2 &= p_1 \cdot \tfrac{364}{365} = 1 \cdot \tfrac{364}{365} = 0.9972\ldots \\
p_3 &= p_2 \cdot \tfrac{363}{365} = 1 \cdot \tfrac{364}{365} \cdot \tfrac{363}{365} = 0.9917\ldots \\
p_4 &= p_3 \cdot \tfrac{362}{365} = 1 \cdot \tfrac{364}{365} \cdot \tfrac{363}{365} \cdot \tfrac{362}{365} = 0.9836\ldots \\
&\vdots
\end{aligned}$$

Fødselsdagsparadokset

Hvis man udregner disse udtryk (evt. via at skrive et program, der gør det), ses at

$$p_{22} = 0.5243\ldots$$
$$p_{23} = 0.4927\ldots$$

Vi var interesserede i, for hvilket n det var mere sandsynligt, at nogen havde samme fødselsdags, end at ingen havde.

Dvs. for hvilket n at $p_n \leq 1/2$.

Af beregningen ovenfor får vi vores svar: allerede når n bliver 23.

Navnet "fødselsdagsparadokset" bruges, fordi dette er et meget mindre antal personer end de fleste intuitivt vil gætte på.

Af samme slags beregninger fås f.eks. $p_{50} = 0.0296\ldots$. Dvs. at der er under 3% sandsynlighed for, at der i en gruppe af 50 personer ikke er to med samme fødselsdato.

Fødselsdagsparadokset og hashing

Recall: At samle personer med tilfældige fødselsdage svarer til at indsætte tilfældigt i en hash-tabel af størrelse 365 (antal dage i et år).

Ovenstående metode kan derfor bruges til at beregne sandsynligheden for, at kollisioner opstår ved indsættelse af n elementer i en hash-tabel, hvis det antages, at hash-funktionen tildeler tabel-indekser til elementer på en tilfældig måde. Man skal blot udskifte 365 med den aktuelle tabelstørrelse.

Morale: vi kan se af sådanne beregninger, at den første kollision opstår overraskende hurtigt, når der indsættes i hash-tabeller (selv hvis vi finder en hash-funktion, der opfører sig, som vi ønsker os).

Det *er* derfor nødvendigt at have et system til at klare kollisioner (f.eks. via lister, jvf. side 23 fra tidligere) for at kunne bruge hashing.

Opsummering om hashing

Vi har set:

- Hovedideen i hashing: Udnyt at heltal kan bruges som indekser til hukommelsesceller.
- Vi har brug for en funktion til at mappe store heltal (ID'er) til små heltal for ikke at spilde plads (vi vil gerne kunne vælge $k \approx n$).
- Det er nødvendigt at have et system til at klare kollisioner.

Vi har ikke set:

- Hvordan man i praksis vælger en hash-funktion, som må forventes for de fleste datasæt at lave få kollisioner.
- Hvordan man laver hash-funktioner på IDs, som ikke er heltal (f.eks. strenge eller generelle objekter).
- Andre systemer (end lister) til at klare kollisioner.
- Andre anvendelser af hash-funktioner (end til at implementere interfacet Random Access): fingerprinting, randomisering af data, kryptografiske anvendelser.

Opsummering om hashing

I praksis ses ofte en todelt hash-funktion:

- En separat del (`hash()` i Python, `hashCode()` i Java) som tager en vilkårlig ID og omformer den til et 64 bit heltal.
- En del indbygget i random access datastrukturen (dictionaries i Python, `HashMap` i Java) som reducerer til et heltal mindre end k.

Med en (evt. todelt) hash-funktion, der er valgt med omhu (som i Python og Java), fungerer hashing meget effektivt i praksis: de fleste kald til datastrukturen vil tage $O(1)$ tid.

Navnet "hashing"? Opstod som en reference til den normale engelske betydning af ordet ("chop and mix"), ved at en typisk hash-funktion sås som at klippe input (et heltal) op i mange bidder, der derefter sættes sammen igen på en kompleks måde for at give output (et heltal), i et forsøg på at sikre at typiske inputsekvenser bliver jævnt fordelt i hash-tabellen.