Søgning

Søgning er et af de problemer, der i forskellige iklædninger optræder igen og igen i praktiske programmeringsopgaver.

Formålet med denne note er at vise eksempler på søgning i forskellige datastrukturer, for til sidst at illustrere, at man én gang for alle kan formulere en generel søgealgoritme, som med et minimum af arbejdsindsats kan forfines til en konkret algoritme.

I det første afsnit gives eksempler på lineær søgning, i det andet på binær søgning, og til sidst beskrives den abstrakte algoritme.

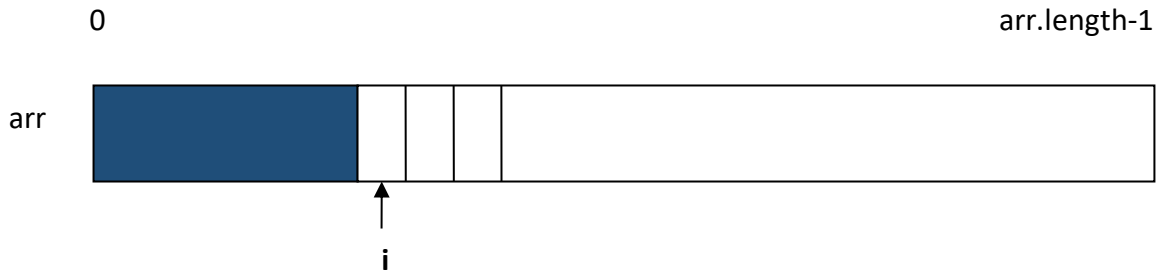
1 Lineær søgning

1.1 Lineær søgning i ikke-sorteret array

I dette eksempel skal vi se, hvorledes en lineær søgning i et usorteret array af `int`, kan programmeres. Vi ønsker at vide, om det søgte element (target) findes i array'et.
Den konkrete metode bliver som følger:

```
public boolean linearSearchArray(int[] arr, int target) {
    boolean found = false;
    int i = 0;
    while (!found && i < arr.length) {
        int k = arr[i];
        if (k == target)
            found = true;
        else
            i++;
    }
    return found;
}
```

Algoritmen kan visualiseres med følgende tegning:



I en lineær søgning undersøges, om det element man er kommet til, er det, der søges efter (om elementet på plads nr. **i** er det ønskede element). Hvis det ønskede element er fundet, stopper søgningen, ellers tælles **i** op, og søgningen fortsætter med det næste element i rækken.

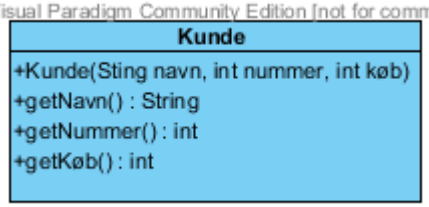
1.2 Lineær søgning i ikke-sorteret ArrayList

En lineær søgning i en ArrayList<String>. Her ønsker vi at kende indekset for det søgte element. Hvis elementet ikke findes, skal -1 returneres.

```
public int linearSearchList(ArrayList<String> list, String target) {
    int indeks = -1;
    int i = 0;
    while (indeks == -1 && i < list.size()) {
        String k = list.get(i);
        if (k.equals(target))
            indeks = i;
        else
            i++;
    }
    return indeks;
}
```

1.3 Lineær søgning i ikke-sorteret ArrayList med Kunde objekter

I dette eksempel vil vi vise, hvordan vi søger i en `ArrayList`, der indeholder referencer til objekter af typen Kunde. Klassen Kunde er beskrevet i nedenstående klassediagram.



Vi ønsker at vide om en kunde med et givet navn findes i listen. Resultatet af søgningen skal være en reference til den søgte kunde (eller null, hvis kunden ikke findes i listen). Hvis der findes flere kunder med det givne navn, er det den første af disse, der returneres.

```
public Kunde linearSearchKunde(ArrayList<Kunde> list, String name) {
    Kunde kunde = null;
    int i = 0;
    while (kunde == null && i < list.size()) {
        Kunde k = list.get(i);
        if (k.getNavn().equals(name))
            kunde = k;
        else
            i++;
    }
    return kunde;
}
```

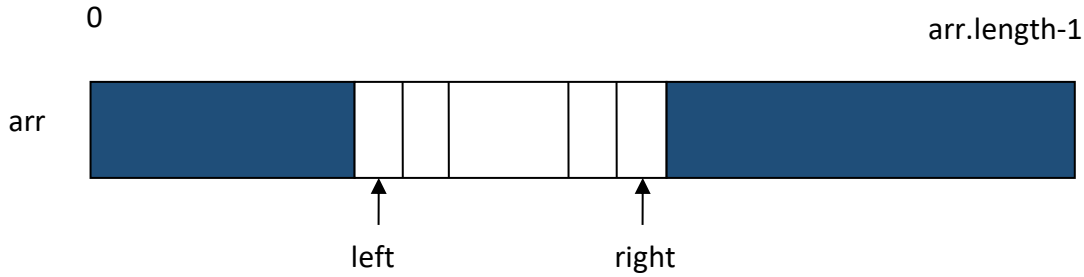
2 Binær søgning

2.1 Binær søgning i sorteret array

I dette afsnit vil vi lave en binær søgning i et *sorteret* array af heltal.

Idéen i binær søgning er, at man starter med at se på værdien af det midterste tal i arrayet. Hvis dette tal er mindre end det, der søges efter, ledes kun videre i højre halvdel af arrayet. Hvis tallet derimod er større end det, der ledes efter, kan vi nøjes med at lede videre i den venstre halvdel af arrayet. Dernæst undersøges det midterste tal, i den halvdel der skulle søges videre i. Sådan fortsættes, indtil tallet enten findes eller der ikke er flere tal at søge blandt.

En binær søgning kræver, at de data, der skal søges i, er sorteret. Herunder vises det, hvordan vi programmerer en binær søgning i et array af heltal. Søgningen skal returnere indekset, hvor elementet findes i arrayet. Hvis elementet ikke findes, returneres -1.
Vi anvender to heltalsvariabler, *left* og *right*, til at repræsentere de data, vi mangler at søge igennem.




```
public int binarySearchArray(int[] arr, int target) {
    int indeks = -1;
    int left = 0;
    int right = arr.length-1;
    while (indeks == -1 && left <= right) {
        int middle = (left + right) / 2;
        int k = arr[middle];
        if (k == target)
            indeks = middle;
        else if (k > target)
            right = middle - 1;
        else
            left = middle + 1;
    }
    return indeks;
}
```

Bemærk: Hvis der søges efter en værdi, som forekommer flere gange i arrayet, finder metoden ikke nødvendigvis den første af disse. Værdien, der returneres, afhænger af, hvilket af de ens tal *middle* først peger på. Hvis vi i et array leder efter tallet 5 og array'et indeholder tallene [2, 5, 5, 5,

10, 12, 20, 28, 28, 30, 36, 50, 54, 60, 72], finder binær søgning det sidste 5-tal; men hvis den indeholder tallene [2, 5, 5, 5, 10, 12, 20, 28, 28], finder binær søgning det første 5-tal.

2.2 Binær søgning i sorteret ArrayList med Kunde objekter

I dette eksempel laver vi en søgning som i afsnit 1.3. Vi antager dog, at listen af kunder er sorteret på kundernes navne, og vi søger efter en kunde med et bestemt navn. Da navnene er af typen `String`, skal sammenligningen i den binære søgning foregå under anvendelse af metoden compareTo() fra String klassen.

```
public Kunde binarySearchKunde(ArrayList<Kunde> list, String name) {
    Kunde kunde = null;
    int left = 0;
    int right = list.size() - 1;
    while (kunde == null && left <= right) {
        int middle = (left+right) / 2;
        Kunde k = list.get(middle);
        if (k.getNavn().compareTo(name) == 0)
            kunde = k;
        else if (k.getNavn().compareTo(name) > 0)
            right = middle - 1;
        else
            left = middle + 1;
    }
    return kunde;
}
```

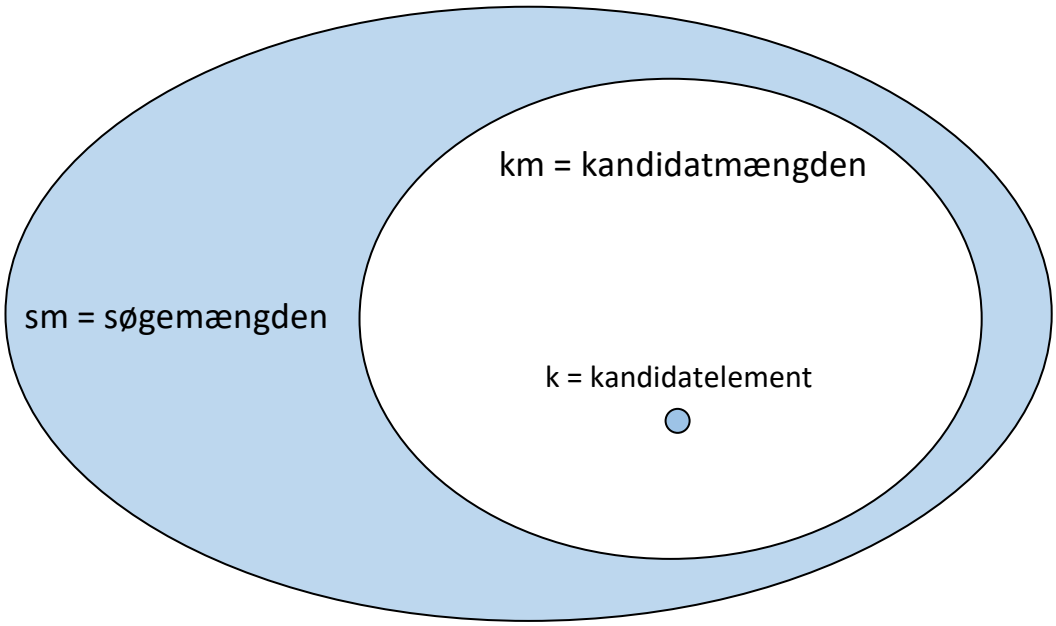
3 Abstrakt søgealgoritme

3.1 Abstrakt algoritme

Enhver søgning er karakteriseret ved, at man blandt en samling af elementer leder efter et eller flere med bestemte karakteristika. Samlingen af elementer, der søges i, kalder vi *søgemængden*, og det element vi leder efter, kalder vi *målelementet*. Vi antager yderligere at vi, et for et, kan inspicere elementerne i søgemængden.

Strategien går da ud på at inspicere søgemængdens elementer et efter et, indtil en forekomst af målelementet er fundet, eller det kan konkluderes, at en forekomst ikke findes.

Strategien kan beskrives ved hjælp af nedenstående figur, hvor *sm* er søgemængden, *km* er *kandidatmængden* - den del af søgemængden der endnu ikke er afsøgt - og *k* er *kandidatelementet* - det element, der i øjeblikket inspiceres.



Ved start er kandidatmængden hele søgemængden. Ved hvert trin i søgningen vælger vi et element i kandidatmængden, kandidatelementet. Hvis kandidatelementet er lig med målelementet, har vi fundet det vi leder efter og er færdig. Hvis ikke, indskrænker vi kandidatmængden (med mindst kandidatelementet). Vi udelukker gerne flere elementer fra kandidatmængden, hvis vi er sikre på, at målelementet ikke er blandt disse elementer.

Ovenstående strategi kan beskrives mere præcist i form af nedenstående abstrakte søgealgoritme (M er målelementet). Resultatet af søgningen returneres i en variabel kaldet resultat:

```
<INITIALISER RESULTAT TIL IKKE-FUNDET>
<INITIALISER KM>
while (<RESULTAT ER IKKE-FUNDET> && <KM ≠ ∅>) {
     <UDVÆLG K FRA KM>
     if (<K ER LIG M>) {
          <SÆT RESULTAT (M ER FUNDET)>
     else
          <SPLIT KM I FORHOLD TIL K OG M>
}
```

For at kunne anvende ovenstående abstrakte søgealgoritme, må vi finde ud af hvordan de ubekendte i algoritmen (angivet med < . . >) skal implementeres i en given situation.

3.2 Konkretiseringer af den abstrakte søgealgoritme

De søgninger der er lavet tidligere i denne note, er alle konkretiseringer af den abstrakte søgealgoritme. I afsnit 1.2 lavede vi en lineær søgning efter et tal i en ikke-sorteret ArrayList.

Herunder sammenlignes den konkrete algoritme med den abstrakte søgealgoritme:

```
public int linearSearchList(ArrayList<String> list, String target) {
        int indeks = -1;
        int i = 0;
        while (indeks == -1 &&
                i < list.size()) {
            String k = list.get(i);
            if (k.equals(target))
                indeks = i;
            else
                i++;
        }
        return indeks;
    }
```

```
<initialiser RESULTAT til ikke-fundet>
<initialiser KM>
while (<RESULTAT er ikke-fundet> &&
          <KM ≠ ∅>) {
     <Udvælg K fra KM>
     if (<K er lig M>)
          <sæt RESULTAT (M er fundet)>
     else
          <split KM i forhold til K og M>
}
```

Ved lineær søgning indskrænkes kandidatmængden kun med ét element ved hvert gennemløb af løkken.
Ved binær søgning indskrænkes kandidatmængden til cirka det halve for hvert gennemløb af løkken.

Samtlige af vore søgninger kan formuleres ud fra søgeskabelonen. Når først vi har lært skabelonen at kende, er det forholdsvis let at anvende skabelonen og dermed få lavet en søgning, hvor risikoen for fejl i algoritmen er lille.

4 Søgning i filer

I dette eksempel skal vi vise, hvordan søgning efter et ord i en fil kan formuleres som en konkretisering af søgeskabelonen. Filen indeholder ét ord på hver linje.

Herunder oversættes den abstrakte søgealgoritme til en konkret algoritme:

```
<initialiser RESULTAT til ikke-fundet>
<initialiser KM>
while (<RESULTAT er ikke-fundet> &&
        <KM ≠ ∅>) {
    <Udvælg K fra KM>
    if (<K er lig M>)
        <sæt RESULTAT (M er fundet)>
    else
        <split KM i forhold til K og M>
}
```

```
public boolean linearSearchFile(
      FileInputStream filein, String word) {
   boolean found = false;
   try (Scanner scan = new Scanner(filein)) {
       while (!found &&
                 scan.hasNext()) {
           String k = scan.nextLine();
           if (k.equals(word))
               found = true;
         //else
         //    KM formindskes af scan.nextLine()
       }
    }
    return found;
}
```

Ved denne lineære søgning indskrænkes kandidatmængden med en linje i tekstfilen ved hvert gennemløb af løkken.

For overblikkets skyld gentages algoritmen her:

```
public boolean linearSearchFile(FileInputStream filein, String word) {
    boolean found = false;
    try (Scanner scan = new Scanner(filein)) {
        while (!found && scan.hasNext()) {
            String k = scan.nextLine();
            if (k.equals(word)) {
                found = true;
            }
        }
    }
    return found;
}
```

5 Alternativ formulering af søgning

I dette afsnit vises en anden ofte anvendt formulering af søgealgoritmen. Princippet i denne formulering er, at metoden returnerer i samme øjeblik, det eftersøgte element er fundet. I eksemplet herunder er vist, hvordan eksemplet fra afsnit 1.2 ser ud med den alternative formulering.

```
public int linearSearchList(ArrayList<String> list, String target) {
    int i = 0;
    while (i < list.size()) {
        String k = list.get(i);
        if (k.equals(target)) {
            return i;
        }
        i++;
    }
    return -1;
}
```

Koden indeholder ingen hjælpevariabel (variablen indeks i afsnit 1.2) til at huske resultatet. Betingelsen i while-sætningen antyder, at hele ArrayList'en løbes igennem, men inde i if-sætningen returneres, når resultatet er fundet. Altså afbrydes gennemløbet, når det eftersøgte er fundet.
Metoden kan returnere på 2 måder: Enten fordi det eftersøgte findes, eller fordi hele ArrayList'en er gennemløbet uden at finde det eftersøgte element.

En del programmører synes, at metoden ikke er så let at læse (at det er grim kode!). Dels antyder betingelsen i while-sætningen, at hele ArrayList'en løbes igennem. Dels har metoden 2 måder at returnere på.
Fortalere for den alternative formulering fremhæver, at de synes, koden er lettere at forstå: dels behøves der ingen hjælpevariabel, dels består betingelsen i while-sætningen ikke af to betingelser sammensat med &&.
Hvis metoden er på få linjer (som i dette eksempel), er den alternative formulering acceptabel/god.

Formuleret som abstrakt søgealgoritme ser den alternative formulering af algoritmen således ud:

```
<INITIALISER KM>
while (<KM ≠ ∅>) {
    <UDVÆLG K FRA KM>
    if (<K ER LIG M>) {
        return <ET RESULTAT (M ER FUNDET)>
    }
    <SPLIT KM I FORHOLD TIL K OG M>
}
return <ET RESULTAT (M ER IKKE FUNDET)
```

6 Avanceret søgning: Strengsøgning

I dette eksempel skal vi skrive en metode, der returnerer det indeks, hvor en tekststreng *m* findes som delstreng i en anden tekststreng *sm*. Metodens hoved bliver således:

```
public int find(String sm, String m)
```

Ved hjælp af Javas strengoperationer kan algoritmen skrives som en helt ren konkretisering af søgeskabelonen. Vi vil imidlertid udlede en algoritme, der ikke passer helt som fod i hose til søgeskabelonen. Pointen er at illustrere, at idéen fra den abstrakte søgealgoritme også kan benyttes i sådanne tilfælde.

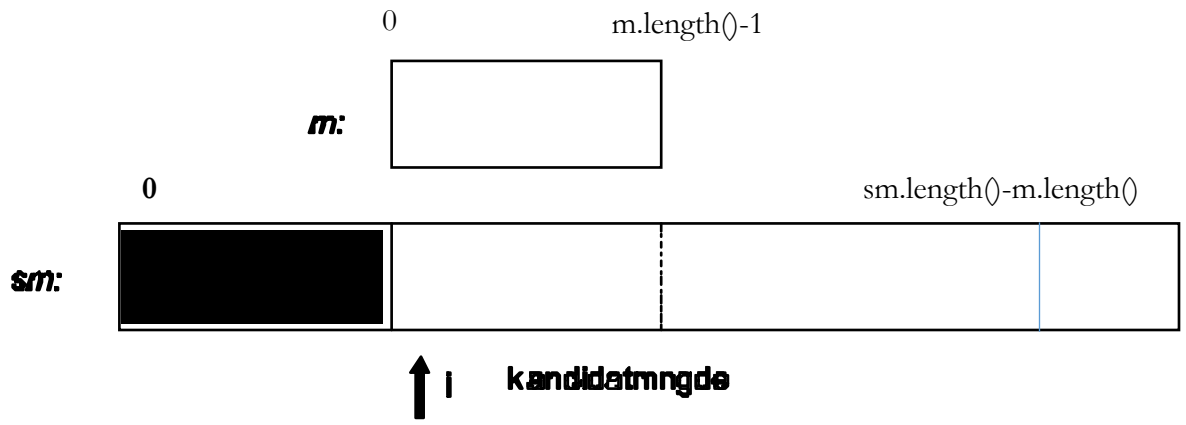
En streng *m* siges at forekomme på position `i` i en streng *sm*, hvis

$$m[j] = sm[i + j] \text{ for } 0 \leq j \leq length(m)-1$$

F.eks. med `sm = "God efterårsferie"` og `m = "efter"`, vil `m` forekomme på position i = 4 i strengen `sm`.

Vi skal således skrive en algoritme, der afgør, om der findes en position `i`, så `m` forekommer på position `i` i `sm`. Til det formål vil vi anvende lineær søgning blandt de mulige positioner i `sm`. Den første mulige position er position 0, og den sidste mulige er position `length(sm)-length(m)`. I eksemplet ovenfor vil den sidste plads, som m kan forekomme på være, 17 - 5 = 12. Vi anvender en heltalsvariabel `i` til at udpege kandidatmængden (mængden af mulige delstrenge), der kan beskrives som:

```
km = sm[i..i+m.length()-1] for 0 <= i <= sm.length()-m.length()
```



Figur: Søgestrategi i strengsøgning.

For hver position *i* vil vi undersøge, om *m* matcher den delstreng, der starter på position *i* i *sm*. Til det formål introduceres en ny metode

```
boolean match(String m, String sm, int i)
```

der returnerer *true*, hvis *m* findes i *sm* fra position *i* og frem.

Hermed bliver den konkrete algoritme som følger:

```
public int find(String sm, String m) {
    int indeks = -1;
    int i = 0;
    while (indeks == -1 && i <= sm.length() - m.length()) {
        if (match(m, sm, i))
            indeks = i;
        else
            i++;
    }
    return indeks;
}
```

Programmeringen af metoden match() sker også ved hjælp af søgeskabelonen – der søges efter *forskellige* bogstaver i m og sm. Metoden ser ud som følger:

```
// Returner om m er en del af sm, startende på indeks i.
private boolean match(String m, String sm, int i) {
    boolean foundDiff = false;
    int j = 0;
    while (!foundDiff && j < m.length()) {
        char k = sm.charAt(i + j);
        if (k != m.charAt(j))
            foundDiff = true;
        else
            j++;
    }
    return !foundDiff;
}
```