EDA016 Programmeringsteknik för D

Läsvecka 12: Algoritmer

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2015

12 Algoritmer

- Att göra denna vecka
- Repetition: Vad är en algoritm?
- Repetition: Min/Max
- Repetition: Linjärsökning
- Algoritmisk tidskomplexitet
- Sortering

Att göra i Vecka 12: Kunna gå från lösningsidé till implementation. Förstå sortering.

1. Läs följande kapitel i kursboken: 7.13, 8.8, 8.9 binärsökning, urvalssortering (selection sort), instickssortering (insertion sort)
2. Gör Övning 11: sortering, objekt
3. Träffas i samarbetsgrupper och hjälp varandra
4. Gör Lab 10: Life

Repetition: Vad är en algoritm?

En algoritm är en stegvis beskrivning av hur man löser ett problem.
Exempel: Min/Max, Linjärsökning, Registrering

Problemlösningsprocessens olika steg (inte nödvändigtvis i denna ordning):

1. identifiera (del)**problemet**:
   exempel: hitta minsta talet
2. Kom på en **lösningsidé**: (kan vara mycket klurigt och svårt)
   exempel: iterera över talen och håll reda på ”minst hittills”
3. Formulera en **stegvis beskrivning** som löser problemet:
   exempel: pseudo-kod med sekvens av instruktioner
4. Implementera en **körbar lösning** i ”riktig” kod:
   exempel: en Java-metod i en klass

Övning: Ge exempel per steg ovan för linjärsökning och registrering.
Det krävs ofta **kreativitiet** i stegen ovan – även i att **känna igen** problemet:

Exempel: skapa highscore-lista kräver dellösningen att hitta *största* talet som är en variant av problemet ”hitta minsta talet” som jag vet hur man kan lösa.

Det finns ofta flera olika sätt (ide, lösning, kod)

Alternativ 1: pseudo-kod ”hitta minsta talet”

```
minSoFar = ett tal STÖRRE än alla andra tal
while (finns fler tal)
    x = nästa tal
    if (x < minSoFar)
        minSoFar = x
return minSoFar
```

Alternativ 2: pseudo-kod ”hitta minsta talet”

```
x = första talet
minSoFar = x
while (finns fler tal)
    x = nästa tal
    if (x > minSoFar)
        minSoFar = x
return minSoFar
```

Vad händer om det inte finns några tal alls? Kolla alla **specialfall**!

Repetition: Linjärsökning

Problem: Sök upp platsen för första förekomsten av ett givet element i en sekvens av element.
Idé: Gå igenom position för position från början till slut och avbryt om rätt tal hittats.

Pseudo-kod:

```
pos = "platsen för det första elementet";
while ("fler element kvar" &&
       "elementet på plats pos inte är det vi söker") {
    pos = "platsen för nästa element";
}
```

Kolla alla **specialfall**!

- Funkar det för inga element alls?
- Vad händer i början? Funkar det för ett enda element?
- Vad händer i slutet? Råkar vi indexera bortom slutet?

Implementation LinjärSökning – variant 1

```
public class Data {
    private int[] v;
    private int n;  // antalet element

    /* här finns konstruktorer och andra metoder */

    public int find1(int nbr) {
        int i = 0;
        while (i < n && v[i] != nbr) {
            i++;
        }
        return (i < n) ? i : -1;
    }
}
```

Kolla alla **specialfall**!

- Funkar det för inga element alls?
- Vad händer i början? Funkar det för ett enda element?
- Vad händer i slutet? Råkar vi indexera bortom slutet?

Implementation LinjärSökning – variant 2

```
public int find2(int nbr) {
    v[n] = nbr;  // lägg till "vaktpost" i slutet
    int i = 0;
    while (v[i] != nbr) {
        i++;
    }
    return (i < n) ? i : -1;
}
```

Kolla alla **specialfall**!

- Funkar det för inga element alls?
- Vad händer i början? Funkar det för ett enda element?
- Vad händer i slutet? Råkar vi indexera bortom slutet?

Implementation LinjärSökning – variant 3

```
public int find3(int nbr) {
    for (int i = 0; i < n; i++) {
        if (v[i] == nbr) {
            return i;
        }
    }
    return -1;
}
```

Kolla alla **specialfall**!

- Funkar det för inga element alls?
- Vad händer i början? Funkar det för ett enda element?
- Vad händer i slutet? Råkar vi indexera bortom slutet?

Implementation LinjärSökning – variant 4

```
public int find4(int nbr) {
    boolean found = false;
    int i = 0;
    while (!found && i < n) {
        if (nbr == v[i]) {
            found = true;
        }
        else {
            i++;
        }
    }
    return (found) ? i : -1;
}
```

Kolla alla **specialfall**!

- Funkar det för inga element alls?
- Vad händer i början? Funkar det för ett enda element?
- Vad händer i slutet? Råkar vi indexera bortom slutet?

Binärsökning i sorterad sekvens

Idé: Om sekvensen är sorterad kan vi utnyttja detta för en mer effektiv sökning, genom att jämföra med mittersta värdet och se om det vi söker finns före eller efter detta, och upprepa med ”halverad” sekvens tills funnet.

Binärsökning i sorterad sekvens

Idé: Om sekvensen är sorterad kan vi utnyttja detta för en mer effektiv sökning, genom att jämföra med mittersta värdet och se om det vi söker finns före eller efter detta, och upprepa med ”halverad” sekvens tills funnet.
Pseudo-kod:

```
    found = false
    while ("finns fler kvar" && !found) {
        mid = "ta reda på mittpunkten i intervallet"
        if (v[mid] == nbr) {
            found = true
        } else if (v[mid] < nbr) {
            "flytta intervallets undre gräns"
        } else {
            "flytta intervallets övre gräns"
        }
    }
    if (found) return mid
    else return "platsen där vi borde stoppa in det saknade elementet"
}
```

Binärsökning i sorterad sekvens

Implementation

```
public int binarySearch(int nbr) {
    int low = 0;       // undre gräns
    int high = n - 1; // övre gräns
    int mid = -1;      // mittpunkt
    boolean found = false;
    while (low <= high && ! found) {
        mid = (low + high) / 2;
        if (v[mid] == nbr) {
            found = true;
        } else if (v[mid] < nbr) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return (found) ? mid : -(low + 1);
}
```

JDK: Arrays.binarySearch(int[] a, int fromIndex, int toIndex, int key)

Algoritmisk komplexitet

Olika algoritmer som löser samma problem kan vara olika effektiva vad gäller

- hur lång tid de tar (tidskopmplexitet) eller
- hur mycket minne de tar (minneskomplexitet).

Genom att studera hur mycket *längre* tid det tar om man *ökar* antalet element, från till exempel n till $10n$ kan man se hur tidsåtgången växer.

Man använder notationen $O(n)$ som uttalas ”ordo n” för att säga att tidsåtgången ökar linjär i förhållande till en ökning av antalet element, medan man skriver $O(n^2)$ om tidsåtgången ökar kvadratiskt i förhållande till en ökning av antalet element.

En for-sats nästlad innuti en for-sats ger typiskt tidskomplexiteten $O(n^2)$.

Tidskomplexitet, sökning

Algoritmteoretisk analys av sökalgoritmerna ger:

- Linjärsökning: $O(n)$
- Binärsökning: $O(\log n)$

Vi har en vektor med 1000 element. Vi har mätt tiden för att söka upp ett element många gånger och funnit att det tar ungefär 1 μs både med linjärsökning och binärsökning. Hur lång tid tar det om vi har fler element i vektorn?

	1,000	10,000	100,000	1,000,000	10,000,000
linjär	1	10	100	1000	10000
binär	1	1.33	1.67	2.00	2.33

I nya kursen ”Utvärdering av programsystem”, obl. för D1, studerar ni detta empiriskt.

Algoritmisk komplexitet studerar ni analytiskt i kursen EDAF05, obl. för D2.

Sortering

Sorteringsproblemet

Problem: Vi har en osorterad sekvens med heltal. Vi vill ordna denna mängd i en sorterad sekvens från minst till störst.

Sorteringsproblemet

Problem: Vi har en osorterad sekvens med heltal. Vi vill ordna denna mängd i en sorterad sekvens från minst till störst.

En *generalisering* av problement:

Vi har många element och en **ordningsrelation** som säger vad vi menar med att ett element är *mindre än* eller *större än* eller *lika med* ett annat element.

Vi vill lösa problemet att ordna elementen i sekvens så att för varje element på plats i så är efterföljande element på plats $i + 1$ större eller lika med elementet på plats i.