

Ursprungligen gjord av:
Rolf Axelsson

Föreläsning 3

Sökning

Linjär sökning
Likhet: ==, equals
Binär sökning
Comparable, Comparator
Tidskomplexitet

Sökning

- Sökning är processen av att finna ett utvalt målelement inom en grupp av saker, eller att utesluta om målelementet finns överhuvudtaget i gruppen.
- Gruppen av saker brukar man kalla sök-pool (search pool).
- Ofta söker man igenom en lista med element.

| 10 | 5 | 88 | 20 | 54 | 54 | 2 | -8 | 4 | 99 | 1000 | 632 | -44 | 1 | 20 |

Linear search i osorterad lista

Problem: Sök efter ett bestämt värde i en array

Algorithm:

Kontrollera elementen i listan med start på element i position 0.

Om värdet på aktuellt element är det sökta värdet så
returnera elementets position

Returnera -1

Halvkod:

Antag att elementet inte finns – sätt variabeln res till -1

För varje element i listan och så länge res == -1

om element i position i har det sökta värdet så
tilldela res elementets position (värdet på i)

Returnera res

eller

För varje element i listan

om element i position i har det sökta värdet så
returnera elementets position

Returnera -1

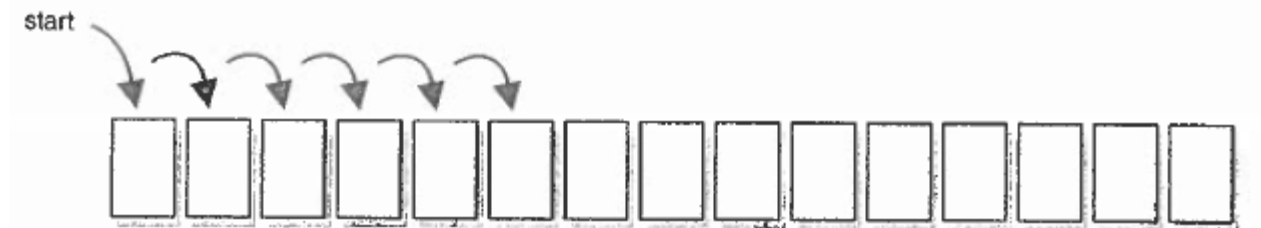


FIGURE 13.1 A linear search

LinearSearch1

==, jämföra enkla variablertyper

Implementering

```
public int indexOf( int[] array, int value ) {  
    int res = -1;  
    for( int i=0; ( i<array.length ) && ( res == -1 ); i++ ) {  
        if( value == array[ i ] ) {  
            res = i;  
        }  
    }  
    return res;  
}
```

eller

```
public int indexOf2( int[] array, int value ) {  
    for( int i=0; i<array.length; i++ ) {  
        if( value == array[ i ] ) {  
            return i;  
        }  
    }  
    return -1;  
}
```

== används när man vill jämföra enkla variablertyper som int, long, double, boolean och char

==, jämföra referensvariabler

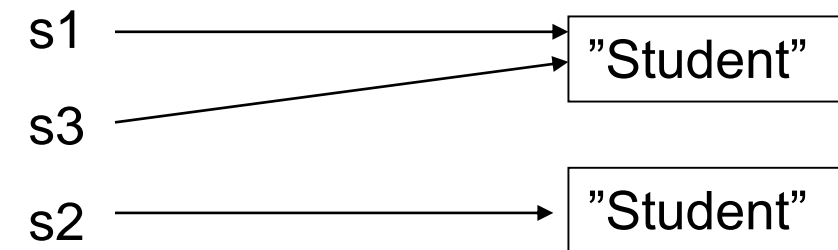
== kan användas vid jämförelse av referensvariabler. Vad man då kontrollerar är om referensvariablerna refererar till samma objekt.

```
String s1 = new String( "Student" ), s2 = new String( "Student" ), s3;  
s3 = s1;  
System.out.println( "s1==s2: " + ( s1 == s2 ) );  
System.out.println( "s1==s3: " + ( s1 == s3 ) );
```

Ger utskrifterna

s1==s2: false

s1==s3: true



Equals

equals, jämföra referensvariabler

- **equals** kan användas vid jämförelse av referensvariabler. I detta fall anropas metoden equals med ett objekt och det andra är argument vid anropet.
- Vad som kontrolleras beror på equals-metoden i klassen. Om klassen inte innehåller en egen version av equals används den ärvda versionen.
- Klassen **Object** implementerar equals-metoden. Den fungerar på samma sätt som ==.

```
String s1 = new String( "Student" ), s2 = new String( "Student" ), s3;
```

```
s3 = s1;
```

```
System.out.println( "s1.equals(s2): " + s1.equals( s2 ) );
```

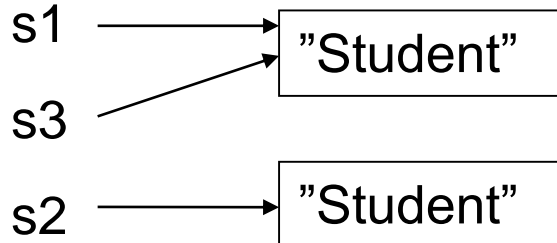
```
System.out.println( "s1.equals(s3): "
```

```
Klassen Object  
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

Ger utskrifterna

```
s1.equals(s2): true
```

```
s1.equals(s3): true
```



```
Klassen String (ungefärlig kod)  
public boolean equals(Object anObject) {  
    if (this == anObject) {  
        return true;  
    }  
    if (anObject instanceof String) {  
        return sameChars(anObject);  
    }  
    return false;  
}
```

Metoden equals överskuggas i RealNbr

```
public class RealNbr {  
    private double value;  
  
    public RealNbr(double value) {  
        this.value = value;  
    }  
  
    public boolean equals(Object obj) {  
        boolean res = (this==obj);  
        if( !res && (obj instanceof RealNbr) ) {  
            RealNbr t = ( RealNbr ) obj;  
            res = ( this.value == t.value );  
        }  
        return res;  
    }  
}
```

RealNbr

LinearSearch2

Binary search – söka i sorterad lista

Att söka i en sorterad lista är ganska enkelt.

Antag att listan är ordnad växande: 11, 14, 19, 20, 21, 22, 24, 26, 29, 30

Algoritm:

Så länge (det finns fler element att söka bland) och (elementet inte hittats)

- Om det mittersta elementet är det sökta så
lagra positionen

- Annars om det mittersta elementet är större än det sökta så
upprepa sökningen på den undre halvan av element

- Annars om det mittersta elementet är mindre än det sökta så
upprepa sökningen på den övre halvan av element

Returnera den lagrade positionen (-1 om inget värde påträffats)

Exempel:

1. Sök värdet 25 i {11, 14, 19, 20, 21, 22, 24, 26, 29, 30}: $25 > 21$
2. Sök värdet 25 i {..., 22, 24, 26, 29, 30}: $25 < 26$
3. Sök värdet 25 i {..., 22, 24, ...}: $25 > 22$
4. Sök värdet 25 i {..., 24, ...}: $25 > 24$
5. Returnera -1

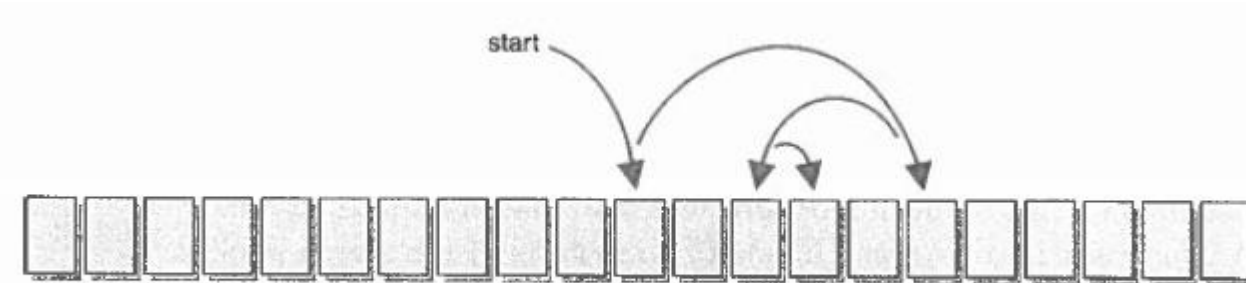


FIGURE 13.2 A binary search

Binary search – söka i sorterad lista

Implementering med **while-loop**:

```
public int binarySearch( int[] array, int key ) {  
    int res = -1, min = 0, max = array.length - 1, pos;  
    while( ( min <= max ) && ( res == -1 ) ) {  
        pos = (min + max) / 2;  
        if( key == array[ pos ] )  
            res = pos;  
        else if( key < array[ pos ] )  
            max = pos - 1;  
        else  
            min = pos + 1;  
    }  
    return res;  
}
```

BinarySearch

Shuffle

Binary search – söka i lista med objekt

Principen när man söker efter objekt i en sorterad lista (`Object[]`) är samma som för enkla variabeltyper. Men för att kunna sortera listan och för att kunna söka i listan krävs något av följande:

- Objekten i listan är naturligt jämförbara (implementerar `Comparable` + `compareTo`)
- En hjälpklass som implementerar `Comparator` och som används vid sortering / sökning.

Klassen `Integer` implementerar `Comparable`:

```
public class Integer implements Comparable<Integer> {  
    ...  
    public int compareTo( Integer obj ) {  
        ...  
    }  
    ...  
}
```

heltal som returvärde:

< 0 mindre än

== samma

> 0 större än

En ordnad array med `Integer`-objekt går att söka binärt.

```
int res = Arrays.binarySearch( integerArray, new Integer( 19 ) );
```

Klassen `RealNbr` implementerar inte `Comparable`. Anropet

```
int res = Arrays.binarySearch( realnbrArray, new RealNbr( 19 ) );
```

ger ett körfel - **`java.lang.ClassCastException: f3.RealNbr ...`**

LISTING 13.1

```

//*****
//  Contact.java      Java Foundations
//
//  Represents a phone contact that implements Comparable.
//*****

public class Contact implements Comparable
{
    private String firstName, lastName, phone;

    //-----
    //  Sets up this contact with the specified information.
    //-----
    public Contact (String first, String last, String telephone)
    {
        firstName = first;
        lastName = last;
        phone = telephone;
    }

    //-----
    //  Returns a string representation of this contact.
    //-----
    public String toString ()
    {
        return lastName + ", " + firstName + ": " + phone;
    }

    //-----
    //  Uses both last and first names to determine lexical ordering.
    //-----
    public int compareTo (Object other)
    {
        int result;

        if (lastName.equals(((Contact)other).lastName))
            result = firstName.compareTo(((Contact)other).firstName);
        else
            result = lastName.compareTo(((Contact)other).lastName);

        return result;
    }
}

```

LISTING 13.2

```

//*****
//  SearchPlayerList.java      Java Foundations
//
//  Demonstrates a linear search of Comparable objects.
//*****

public class SearchPlayerList
{
    //-----
    //  Creates an array of Contact objects, then searches for a
    //  particular player.
    //-----
    public static void main (String[] args)
    {
        Contact[] players = new Contact[7];

        players[0] = new Contact ("Rodger", "Federer", "610-555-7384");
        players[1] = new Contact ("Andy", "Roddick", "215-555-3827");
        players[2] = new Contact ("Maria", "Sharapova", "733-555-2969");
        players[3] = new Contact ("Venus", "Williams", "663-555-3984");
        players[4] = new Contact ("Lleyton", "Hewitt", "464-555-3489");
        players[5] = new Contact ("Eleni", "Daniilidou", "322-555-2284");
        players[6] = new Contact ("Serena", "Williams", "243-555-2837");

        Contact target = new Contact ("Eleni", "Daniilidou", "");

        Contact found = (Contact)Searching.linearSearch(players, target);

        if (found == null)
            System.out.println ("Player was not found.");
        else
            System.out.println ("Found: " + found);
    }
}

```

OUTPUT

Found: Daniilidou, Eleni: 322-555-2284

Binary search – söka i lista med objekt

Alla klasser implementerar inte Comparable.

Och även om en klass implementerar Comparable så ordnas kanske objekten enligt fel princip (felaktig sorteringsordning).

Om man önskar sortera en array med RealNbr avtagande så duger inte den naturliga sorteringsordningen (växande).

Men det går bra att skicka med ytterligare ett argument till sort-metoden:

```
Arrays.sort( realNbrArray, new Decrease() );
```

där det andra argumentet ska vara en klass som implementerar Comparator.

```
public interface Comparator<T> {  
    public int compare( T obj1, T obj2 );  
}
```

```
-----  
public class Decrease implements Comparator<RealNbr> {
```

```
    public int compare( RealNbr nbr1, RealNbr nbr2 ) {  
        if( nbr1.getValue() > nbr2.getValue() )    // return -nbr1.compareTo( nbr2 );  
            return -1;  
        else if( nbr1.getValue() == nbr2.getValue() )  
            return 0;  
        else  
            return 1;  
    }  
}
```

Decrease

Binary search – rekursiv

Det går bra att uttrycka binarySearch rekursivt:

```
Om min > max
    returnera -1
pos = ( min + max ) / 2;
om elementet i pos är det sökta
    returnera pos
annars om det sökta elementet < elementet i pos
    returnera sökresultatet i den nedre halvan
annars
    returnera sökresultatet i den övre halvan
```

```
public int binarySearchRek( Object[] array, Object key, Comparator comp, int min, int
max ) {
    if( min > max )
        return -1;
    int pos = (min + max) / 2;
    int res = comp.compare( key, array[ pos ] );
    if( res == 0 )
        return pos;
    else if( res < 0 ) // key före mittenelement i listan
        return binarySearchRek( array, key, comp, min, pos - 1 );
    else // key efter mittenelement i listan
        return binarySearchRek( array, key, comp, pos + 1, max );
}
```

Tidskomplexitet

Tidskomplexiteten hos en algoritm är det antal steg en algoritm måste genomföra (hur lång tid tar den att genomföra).

Exempel: Linear search

```
public int indexOf( int[] array, int value ) {  
    int res = -1;  
    for( int i=0; ( i<array.length ) && ( res == -1 ); i++ ) {  
        if( value == array[ i ] ) {  
            res = i;  
        }  
    }  
    return res;  
}
```

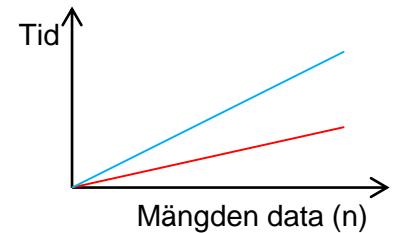
Asymptotisk komplexitet

Vid sökning efter ett speciellt värde i en lista med 9 element så:

- A I bästa faller så finns värdet i första positionen, loopen upprepas 1 gång
- B I sämsta fallet så finns inte värdet i listan, loopen upprepas 9 gånger
- C Om det sökta värdet alltid finns i listan upprepas loopen i genomsnitt 5 gånger

Vid sökning efter ett speciellt värde i en lista med n element så:

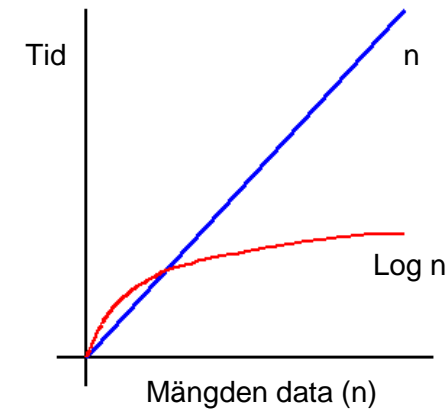
- B I sämsta fallet så finns inte värdet i listan, loopen upprepas n gånger
- C Om det sökta värdet alltid finns i listan upprepas loopen i genomsnitt $n/2$ gånger



Vid analys av tidskomplexitet så är vi endast intresserade av hur antalet steg ändras då antalet element ändras. I båda ovanstående fall ökar antalet steg linjärt.

Tidskomplexitet – binary search

```
public int binarySearch( int[] array, int key ) {  
    int res = -1, min = 0, max = array.length - 1, pos;  
    while( ( min <= max ) && ( res == -1 ) ) {  
        pos = (min + max) / 2;  
        if( key == array[ pos ] )  
            res = pos;  
        else if( key < array[ pos ] )  
            max = pos - 1;  
        else  
            min = pos + 1;  
    }  
    return res;  
}
```



Vid sökning efter ett speciellt värde i en lista med 9 element så:

- A I bästa faller så finns värdet i mittpositionen, loopen upprepas 1 gång
- B I sämsta fallet så finns inte värdet i listan, loopen upprepas 3-4 gånger
- C Om det sökta värdet alltid finns i listan upprepas loopen i genomsnitt ca 3 gånger

Vid sökning efter ett speciellt värde i en lista med n element så:

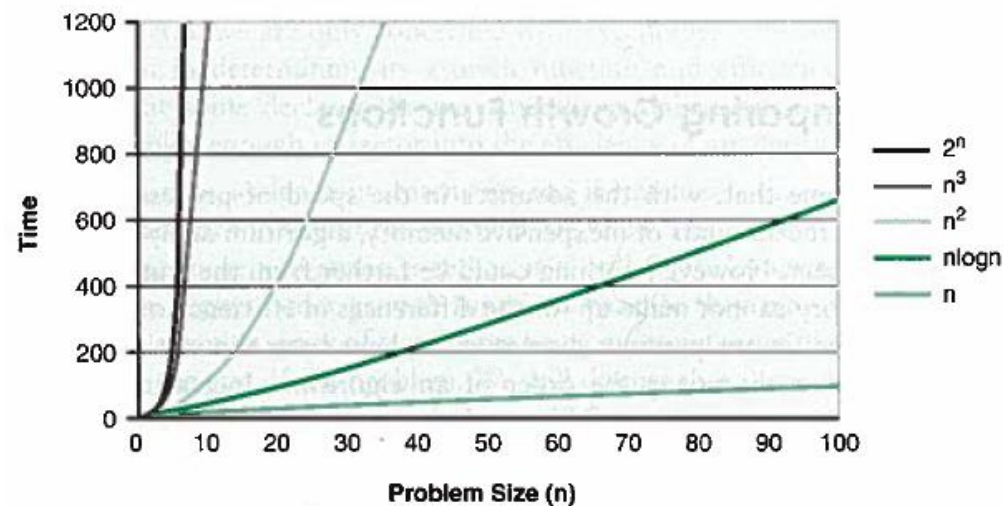
- B I sämsta fallet så finns inte värdet i listan, loopen upprepas $\log n$ gånger
- C Om det sökta värdet alltid finns i listan upprepas loopen i genomsnitt $\log n/2$ gånger

Tidskomplexiteten är då **$O(\log n)$** (uttalas ordo log n)

Tidskomplexitet

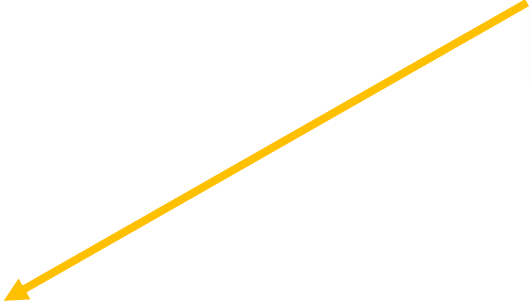
Vid analys av algoritmer få man funktioner som beskriver antalet operationer som krävs för att hantera n element. När man anger tidskomplexiteten förenklar man uttrycket till den dominanta termen.

Funktion	Order	Label
$t(n) = 17$	$O(1)$	Konstant
$t(n) = 3 \log n$	$O(\log n)$	Logaritmisk
$t(n) = 20n - 4$	$O(n)$	Linjär
$t(n) = 12n \cdot \log n + 100n$	$O(n \log n)$	$n \log n$
$t(n) = 3n^2 + 5n - 2$	$O(n^2)$	kvadratisk
$t(n) = 8n^3 + 3n^2$	$O(n^3)$	kubisk
$t(n) = 2^n + 18n^2 + 3n$	$O(2^n)$	exponentiell

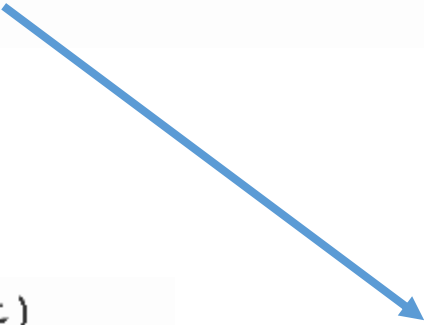


Välj en bättre algoritm istället för en snabbare processor.

```
for (int count = 0; count < n; count++)  
{  
    printsum (count);  
}
```



```
public void printsum(int count)  
{  
    int sum = 0;  
    for (int i = 1; i < count; i++)  
        sum += i;  
    System.out.println (sum);  
}
```


$$\sum_{i=1}^n i = n(n+1)/2$$

```
public void printsum(int count)  
{  
    sum = count * (count+1)/2;  
    System.out.println (sum);  
}
```

Ett exempel

```
public void sample(int n)
{
    printsum(n);                /* this method call is O(1) */

    for (int count = 0; count < n; count++) /* this loop is O(n) */
        printsum (count);

    for (int count = 0; count < n; count++) /* this loop is O(n²)*/
    {
        for (int count2 = 0; count2 < n; count2++)
            System.out.println (count, count2);
    }
}
```

Rekursion

```
// This method returns the sum of 1 to num
public int sum (int num)
{
    int result;
    if (num == 1)
        result = 1;
    else
        result = num + sum (num-1);
    return result;
}
```

Vad är Ordo för den rekursiva funktionen sum?

Frågor?

