Introduction to Programming: Lecture 14

K Narayan Kumar

Chennai Mathematical Institute

http://www.cmi.ac.in/~kumar

01 October 2013

Arrays

- Lists store a collection of elements.

Arrays

- Lists store a collection of elements.
- Accessing the first element of a list takes just one step.
- Accessing the i^{th} element of a list takes i steps.

```
a :: [Int]
...
b = a!!i
...
```

Arrays

- Lists store a collection of elements.
- Accessing the first element of a list takes just one step.
- Accessing the i^{th} element of a list takes i steps.

```
a :: [Int]
...
b = a!!i
...
```

- It is useful to be able to access every element in equal time (constant?)

Arrays

- Lists store a collection of elements.
- Accessing the first element of a list takes just one step.
- Accessing the i^{th} element of a list takes i steps.

```
a :: [Int]
...
b = a!!i
...
```

- It is useful to be able to access every element in equal time (constant?)

  `Array` give one way to do this.

Arrays

- Lists store a collection of elements.
- Accessing the first element of a list takes just one step.
- Accessing the i^{th} element of a list takes i steps.

```
a :: [Int]
...
b = a!!i
...
```

- It is useful to be able to access every element in equal time (constant?)

  `Array` give one way to do this.

  The module `Data.Array` has to be imported to use arrays.

Arrays

- An Example

```
myarray :: Array Int String
```

Arrays

- An Example

  ```
  myarray :: Array Int String
  ```

- This definition says that
  - The indices of the array come from `Int` and
  - The values come from `String`

Arrays

- An Example

```
myarray :: Array Int String
```

- This definition says that
  - The indices of the array come from `Int` and
  - The values come from `String`
- Here is one way to create an array.

```
myarray = listArray (0,2) ["one","two","three"]
```

Arrays

- An Example

```
myarray :: Array Int String
```

- This definition says that
  - The indices of the array come from `Int` and
  - The values come from `String`
- Here is one way to create an array.

```
myarray = listArray (0,2) ["one","two","three"]
```

- The resulting array is

Index	0	1	2
Value	"one"	"two"	"three"

Arrays

- An Example

```
myarray :: Array Int String
```

- This definition says that
  - The indices of the array come from `Int` and
  - The values come from `String`
- Here is one way to create an array.

```
myarray = listArray (0,2) ["one","two","three"]
```

- The resulting array is

Index	0	1	2
Value	"one"	"two"	"three"

- `listArray` takes a range of values from the index type and list of values to create an array.

Arrays

- An Example

```
myarray :: Array Int String
```

- This definition says that
  - The indices of the array come from `Int` and
  - The values come from `String`
- Here is one way to create an array.

```
myarray = listArray (0,2) ["one","two","three"]
```

- The resulting array is

Index	0	1	2
Value	"one"	"two"	"three"

- `listArray` takes a range of values from the index type and list of values to create an array.
- Accessing an element is done using the `!` operator:

```
myarray!1 = "two"
```

Arrays ...

- `Array a b` is a valid type when `a` belongs to `Ix`
- `Ix` is a collection of types that are in `Ord` and further can be enumerated

Arrays ...

- `Array a b` is a valid type when `a` belongs to `Ix`
- `Ix` is a collection of types that are in `Ord` and further can be enumerated
  - If `Ix a` then if `x,y` are of type `a` and `x <= y` then the range of values between `x` and `y` is defined and finite

Arrays ...

- `Array a b` is a valid type when `a` belongs to `Ix`
- `Ix` is a collection of types that are in `Ord` and further can be enumerated
  - If `Ix a` then if `x,y` are of type `a` and `x <= y` then the range of values between `x` and `y` is defined and finite
- `Int`, `Char` are in `Ix` but `Float` is not.

Arrays ...

- `Array a b` is a valid type when `a` belongs to `Ix`
- `Ix` is a collection of types that are in `Ord` and further can be enumerated
  - If `Ix a` then if `x,y` are of type `a` and `x <= y` then the range of values between `x` and `y` is defined and finite
- `Int`, `Char` are in `Ix` but `Float` is not.
- Example:

```
x = listArray ('e','g') ["one","two","three"]
```

Arrays ...

- `Array a b` is a valid type when `a` belongs to `Ix`
- `Ix` is a collection of types that are in `Ord` and further can be enumerated
  - If `Ix a` then if `x,y` are of type `a` and `x <= y` then the range of values between `x` and `y` is defined and finite
- `Int`, `Char` are in `Ix` but `Float` is not.
- Example:

```
x = listArray ('e','g') ["one","two","three"]
```

The resulting array is

Index	'e'	'f'	'g'
Value	"one"	"two"	"three"

More about indices

- Any type `a` that belongs to the typeclass `Ix` must provide the following functions:

```
range :: (a,a) -> [a]
index :: (a,a) a -> Int
inRange :: (a,a) -> a -> Bool
```

More about indices

- Any type `a` that belongs to the typeclass `Ix` must provide the following functions:

```
range :: (a,a) -> [a]
index :: (a,a) a -> Int
inRange :: (a,a) -> a -> Bool
```

- `range` returns the list of elements in the given interval.

```
range (0,5) = [0,1,2,3,4,5]
```

More about indices

- Any type `a` that belongs to the typeclass `Ix` must provide the following functions:

```
range :: (a,a) -> [a]
index :: (a,a) a -> Int
inRange :: (a,a) -> a -> Bool
```

- `range` returns the list of elements in the given interval.

```
range (0,5) = [0,1,2,3,4,5]
```

- `index` returns the position of a given value in the range.

```
index (0,5) 2 = 2
```

More about indices

- Any type `a` that belongs to the typeclass `Ix` must provide the following functions:

```
range :: (a,a) -> [a]
index :: (a,a) a -> Int
inRange :: (a,a) -> a -> Bool
```

- `range` returns the list of elements in the given interval.

```
range (0,5) = [0,1,2,3,4,5]
```

- `index` returns the position of a given value in the range.

```
index (0,5) 2 = 2
```

- `inRange` checks of the value lies within the range.

```
inRange (0,5) 6 = False
```

Creating Arrays

- We may also create an array by supplying the index-value pairs (associative list) as a list.

Creating Arrays

- We may also create an array by supplying the index-value pairs (associative list) as a list.

```
myarray :: Array Int String
myarray =  array (0,2)
                 [(1,"two"), (0,"one"),(2,"three")]
```

Creating Arrays

- We may also create an array by supplying the index-value pairs (associative list) as a list.

```
myarray :: Array Int String
myarray =  array (0,2)
                 [(1,"two"), (0,"one"),(2,"three")]
```

has the same effect as the earlier definition

```
listArray (0,2) ["one", "two", "three"]
```

Creating Arrays

- We may also create an array by supplying the index-value pairs (associative list) as a list.

```
myarray :: Array Int String
myarray =  array (0,2)
                 [(1,"two"), (0,"one"),(2,"three")]
```

has the same effect as the earlier definition

```
listArray (0,2) ["one", "two", "three"]
```

- The `array` function allows you to supply the associative list in any order.

Creating Arrays

- We may also create an array by supplying the index-value pairs (associative list) as a list.

```
myarray :: Array Int String
myarray =  array (0,2)
                 [(1,"two"), (0,"one"),(2,"three")]
```

has the same effect as the earlier definition

```
listArray (0,2) ["one", "two", "three"]
```

- The `array` function allows you to supply the associative list in any order.
- We may also omit some elements.

```
array (0,2) [(0,5),(2,7)]
```

Creating Arrays

- We may also create an array by supplying the index-value pairs (associative list) as a list.

```
myarray :: Array Int String
myarray =  array (0,2)
                 [(1,"two"), (0,"one"),(2,"three")]
```

  has the same effect as the earlier definition

```
listArray (0,2) ["one", "two", "three"]
```

- The `array` function allows you to supply the associative list in any order.
- We may also omit some elements.

```
array (0,2) [(0,5),(2,7)]
```

- Both the functions `array` and `listArray` take time proportional to the range of the array.

Creating Arrays

- We may also create an array by supplying the index-value pairs (associative list) as a list.

```
myarray :: Array Int String
myarray =  array (0,2)
                 [(1,"two"), (0,"one"),(2,"three")]
```

  has the same effect as the earlier definition

```
listArray (0,2) ["one", "two", "three"]
```

- The `array` function allows you to supply the associative list in any order.
- We may also omit some elements.

```
array (0,2) [(0,5),(2,7)]
```

- Both the functions `array` and `listArray` take time proportional to the range of the array. (??)

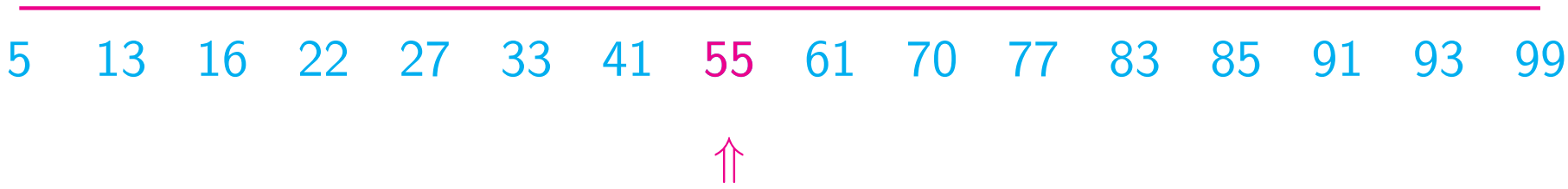
Searching in a sorted array

Searching for 77

5 13 16 22 27 33 41 55 61 70 77 83 85 91 93 99

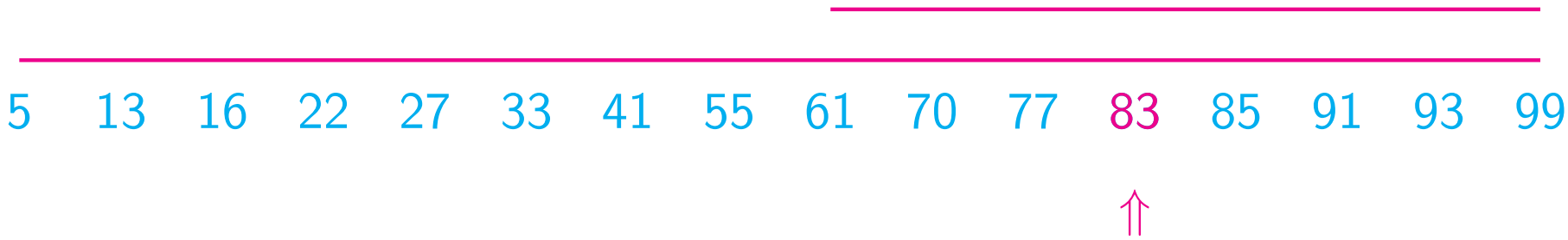
Searching in a sorted array

Searching for 77



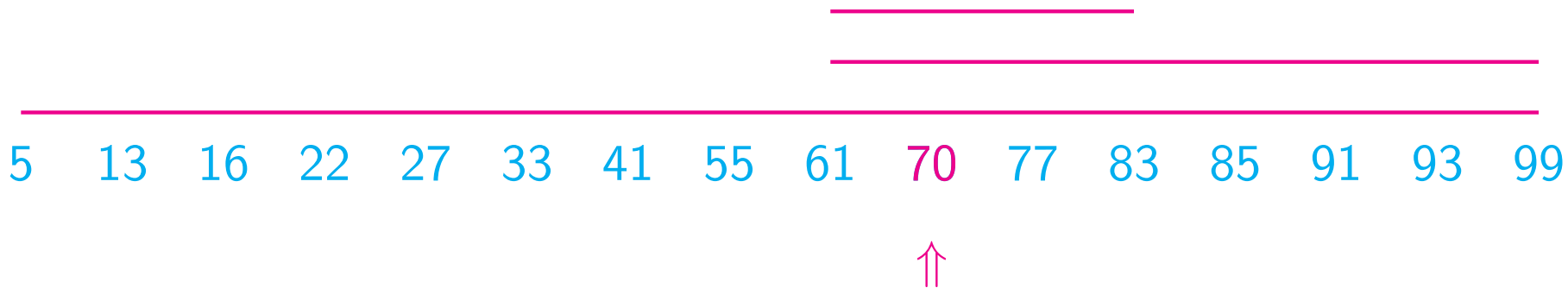
Searching in a sorted array

Searching for 77



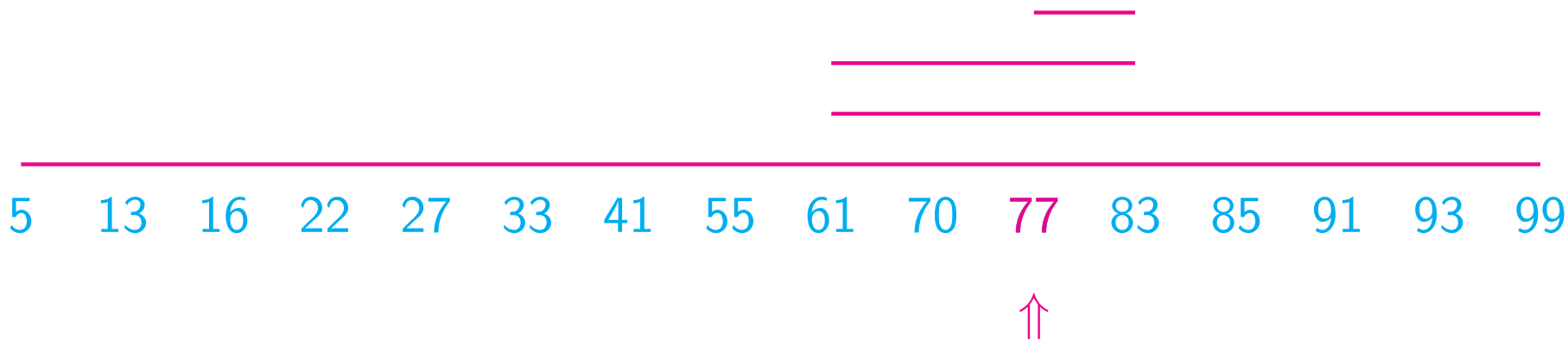
Searching in a sorted array

Searching for 77



Searching in a sorted array

Searching for 77



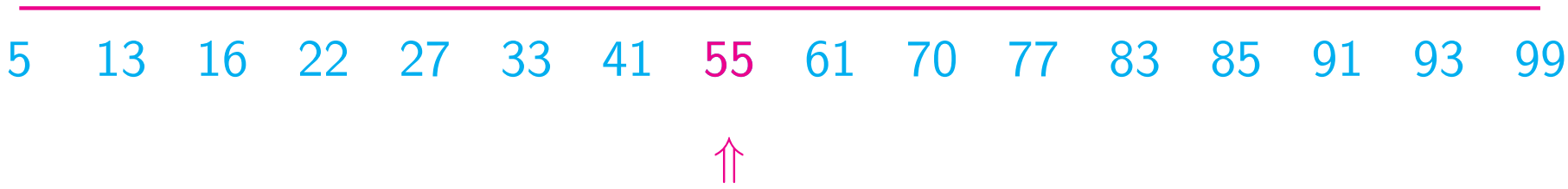
Searching in a sorted array

Searching for 24

5 13 16 22 27 33 41 55 61 70 77 83 85 91 93 99

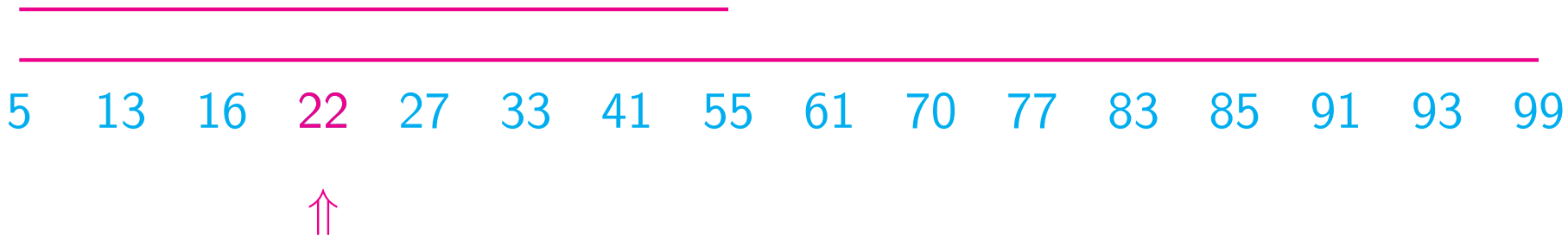
Searching in a sorted array

Searching for 24



Searching in a sorted array

Searching for 24

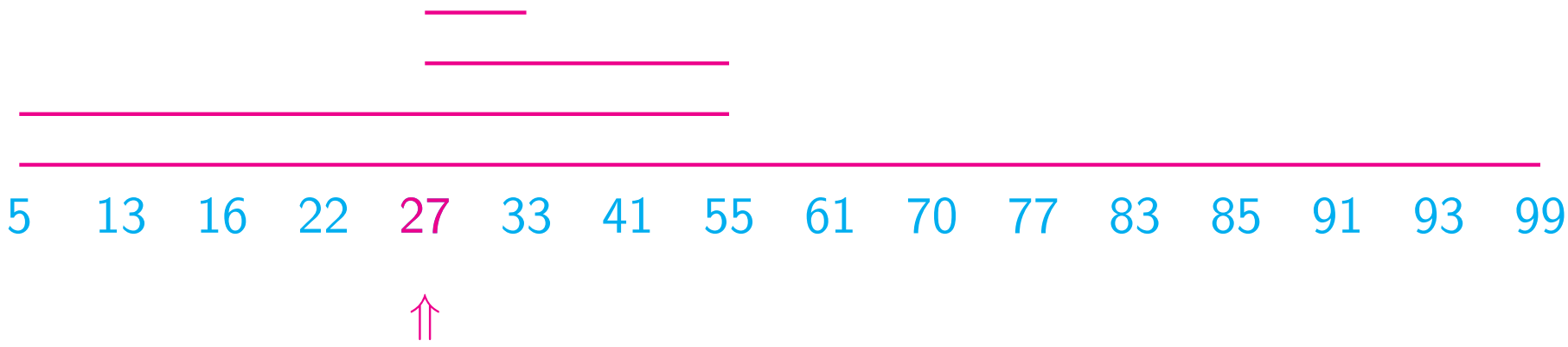


Searching in a sorted array

Searching for 24

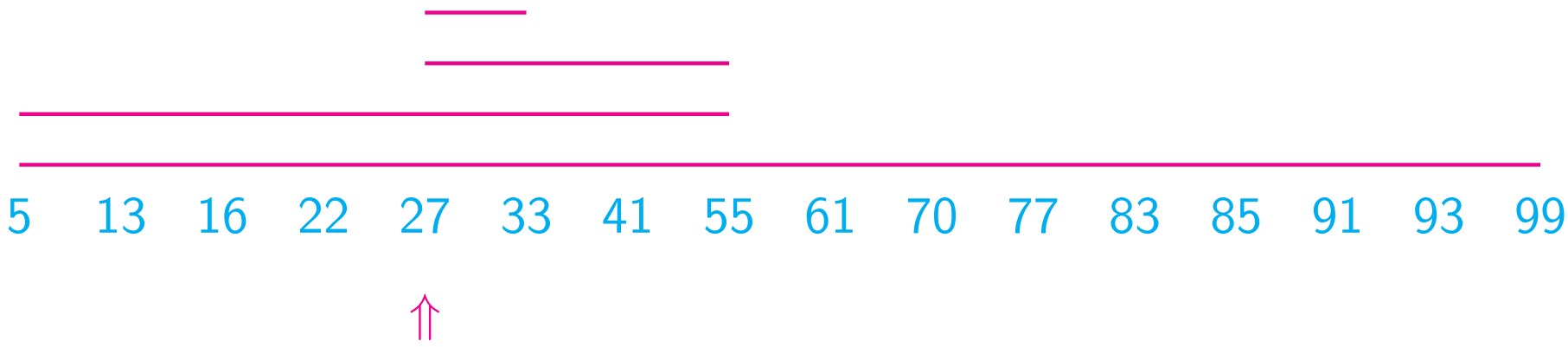
Searching in a sorted array

Searching for 24



Searching in a sorted array

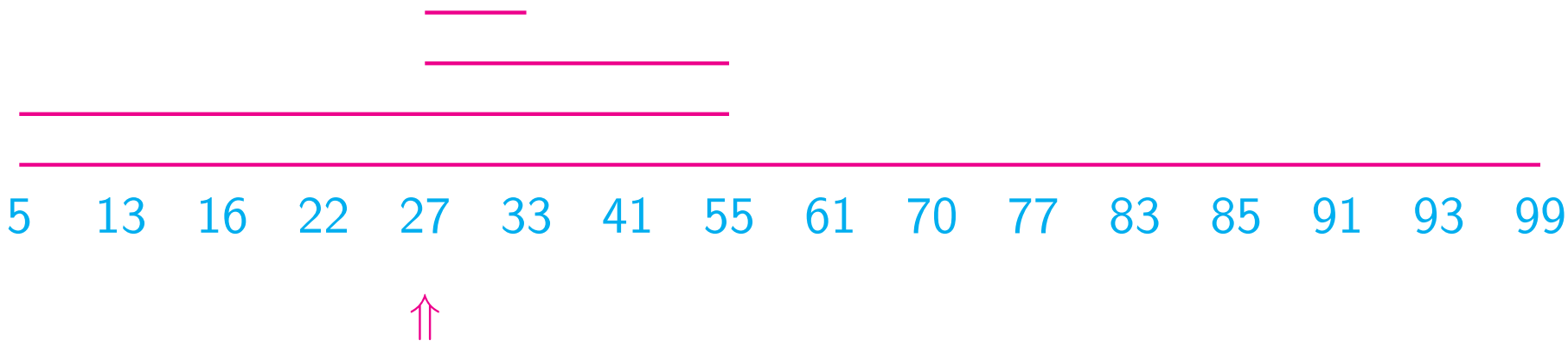
Searching for 24



- Each step halves the interval to search

Searching in a sorted array

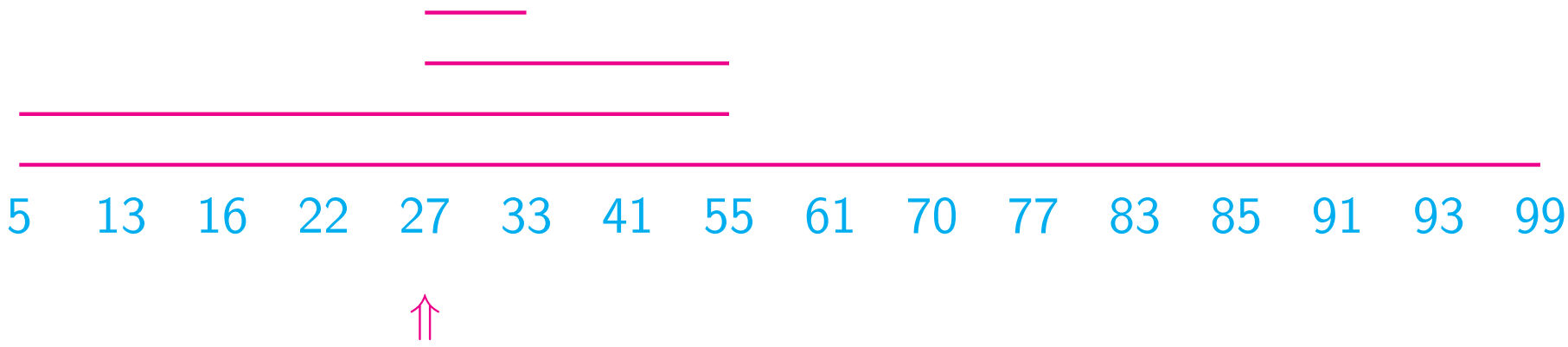
Searching for 24



- Each step halves the interval to search
- Keep halving till we reach an interval of size 1

Searching in a sorted array

Searching for 24



- Each step halves the interval to search
- Keep halving till we reach an interval of size 1
- Searching a sorted list of size N takes $\log_2 N$ steps

Binary Searching in Haskell

- The following function checks if the value `v` appears between positions `b` and `e` in the array `ar`

```
import Data.Array
bAux :: Ord a => (Int,Int) -> a ->
                              Array Int a ->  Bool
bAux (b,e) v ar
  | b > e       = False
  | (ar!m) == v = True
  | (v < ar!m)  = bAux (b,m-1) v ar
  | otherwise   = bAux (m+1,e) v ar
  where
   m = div (b+e) 2
```

Binary Searching in Haskell

- The following function checks if the value `v` appears between positions `b` and `e` in the array `ar`

```
import Data.Array
bAux :: Ord a => (Int,Int) -> a ->
                              Array Int a ->  Bool
bAux (b,e) v ar
  | b > e       = False
  | (ar!m) == v = True
  | (v < ar!m)  = bAux (b,m-1) v ar
  | otherwise   = bAux (m+1,e) v ar
  where
   m = div (b+e) 2
```

- Searching for a value `v` in a sorted array `ar`

```
bsearch :: Ord a => a -> Array Int a -> Bool
bsearch v ar = bAux (bounds ar) v ar
```

Arrays to Lists and Back

- `elems` function converts an array into a list of values.

Arrays to Lists and Back

- `elems` function converts an array into a list of values.
- To convert a list into an `Int` indexed array, we can use

```
listToArray ls = listArray (0,l-1) ls
  where
  l = length ls
```

Searching in an unsorted list

- Given a unsorted list of values on which a number of search queries need to be answered

Searching in an unsorted list

- Given a unsorted list of values on which a number of search queries need to be answered
- Sort the list

Searching in an unsorted list

- Given a unsorted list of values on which a number of search queries need to be answered
- Sort the list using say `mergesort` ...

Searching in an unsorted list

- Given a unsorted list of values on which a number of search queries need to be answered
- Sort the list using say `mergesort` ...
- Transfer the list into array.

Searching in an unsorted list

- Given a unsorted list of values on which a number of search queries need to be answered
- Sort the list using say `mergesort` ...
- Transfer the list into array.
- Use `bsearch` to carry out the searches efficiently.

Functions on Arrays

Functions on Arrays

- The function `bounds` returns the end-points of the range of indices used by the array.

```
bounds myarray = (0,2)
```

Functions on Arrays

- The function `bounds` returns the end-points of the range of indices used by the array.

```
bounds myarray = (0,2)
```

- The function `indices` returns the range defined by its bounds.

```
indices = range.bounds
```

Functions on Arrays

- The function `bounds` returns the end-points of the range of indices used by the array.

  ```
  bounds myarray = (0,2)
  ```

- The function `indices` returns the range defined by its bounds.

  ```
  indices = range.bounds
  ```

- The function `elems` returns the elements of the array.

  ```
  elems ar = [ar!i  |  i <- indices ar]
  ```

  `elems` is undefined if the array value is undefined for any of the indices.

Functions on Arrays

- The function `bounds` returns the end-points of the range of indices used by the array.

  ```
  bounds myarray = (0,2)
  ```

- The function `indices` returns the range defined by its bounds.

  ```
  indices = range.bounds
  ```

- The function `elems` returns the elements of the array.

  ```
  elems ar = [ar!i  |  i <- indices ar]
  ```

  `elems` is undefined if the array value is undefined for any of the indices.

- `assocs` returns the associative list describing the array.

  ```
  assocs myarray =
  [(0,"one"),(1,"two"),(2,"three")]
  ```

"Changing" the values in an array

- Arrays, like all other values in Haskell, are immutable.
  Arrays are essentially functions!

"Changing" the values in an array

- Arrays, like all other values in Haskell, are immutable.
  Arrays are essentially functions!
- We can create an array from another by updating some of the entries.

"Changing" the values in an array

- Arrays, like all other values in Haskell, are immutable.
  Arrays are essentially functions!
- We can create an array from another by updating some of the entries.

  This is done using the // operator and by supplying an associative list of changes

"Changing" the values in an array

- Arrays, like all other values in Haskell, are immutable.
  Arrays are essentially functions!
- We can create an array from another by updating some of the entries.

  This is done using the `//` operator and by supplying an associative list of changes

```
(listArray (0,2) [2,3,4])//([(1,7),(2,8)]
  = listArray (0,2) [2,7,8]
```

"Changing" the values in an array

- Arrays, like all other values in Haskell, are immutable. Arrays are essentially functions!
- We can create an array from another by updating some of the entries.

  This is done using the `//` operator and by supplying an associative list of changes

```
(listArray (0,2) [2,3,4])//([(1,7),(2,8)]
  = listArray (0,2) [2,7,8]
```

- However, note that updating an array using the `//` operator is an expensive operation with cost proportional to the size of the array.

"Changing" the values in an array

- Arrays, like all other values in Haskell, are immutable. Arrays are essentially functions!
- We can create an array from another by updating some of the entries.

  This is done using the `//` operator and by supplying an associative list of changes

```
(listArray (0,2) [2,3,4])//([(1,7),(2,8)]
  = listArray (0,2) [2,7,8]
```

- However, note that updating an array using the `//` operator is an expensive operation with cost proportional to the size of the array.
- Efficiently updatable arrays can be created in Haskell, but that needs additional concepts.

Another array constructing function

- The `accumArray` function takes a "accumulating" function and an associative list and creates an array.

```
accumArray (+) 100 (0,2) [(0,1),(1,3),(2,4)]
= listArray (0,2) [101,103,104]
```

Another array constructing function

- The `accumArray` function takes a "accumulating" function and an associative list and creates an array.

```
accumArray (+) 100 (0,2) [(0,1),(1,3),(2,4)]
= listArray (0,2) [101,103,104]

accumArray (+) 100 (0,2) [(0,1),(1,3),(0,4)]
= listArray (0,2) [105,103,100]
```

Another array constructing function

- The `accumArray` function takes a "accumulating" function and an associative list and creates an array.

```
accumArray (+) 100 (0,2) [(0,1),(1,3),(2,4)]
= listArray (0,2) [101,103,104]

accumArray (+) 100 (0,2) [(0,1),(1,3),(0,4)]
= listArray (0,2) [105,103,100]
```

- The type of `accumArray` is

Another array constructing function

- The `accumArray` function takes a "accumulating" function and an associative list and creates an array.

```
accumArray (+) 100 (0,2) [(0,1),(1,3),(2,4)]
= listArray (0,2) [101,103,104]

accumArray (+) 100 (0,2) [(0,1),(1,3),(0,4)]
= listArray (0,2) [105,103,100]
```

- The type of `accumArray` is

```
accumArray :: Ix i => (a ->  b -> a) -> a ->
                  -> (i,i) -> [(i,b)] -> Array i a
```

Another array constructing function

- The `accumArray` function takes a "accumulating" function and an associative list and creates an array.

```
accumArray (+) 100 (0,2) [(0,1),(1,3),(2,4)]
= listArray (0,2) [101,103,104]

accumArray (+) 100 (0,2) [(0,1),(1,3),(0,4)]
= listArray (0,2) [105,103,100]
```

- The type of `accumArray` is

```
accumArray :: Ix i => (a ->  b -> a) -> a ->
                  -> (i,i) -> [(i,b)] -> Array i a
```

- Also works in linear time on the length of the associative list plus the range.

An old example: `minout`

- `minout :: [Int] -> Int`
  `minout l` is the minimum nonnegative number not in `l` assuming that all elements in `l` are nonnegative and distinct.
  - `minout [3,1,2] = 0`
  - `minout [1,5,3,0,2] = 4`
  - `minout [11,5,3,0] = 1`
- How do we compute `minout`?

An old example: `minout`

- `minout :: [Int] -> Int`
  `minout l` is the minimum nonnegative number not in `l` assuming that all elements in `l` are nonnegative and distinct.
  - `minout [3,1,2] = 0`
  - `minout [1,5,3,0,2] = 4`
  - `minout [11,5,3,0] = 1`
- How do we compute `minout`?
- The linear time solution via lists involved a rather clever divide and conquer algorithm.

An old example: `minout`

- `minout :: [Int] -> Int`
  `minout l` is the minimum nonnegative number not in `l` assuming that all elements in `l` are nonnegative and distinct.
  - `minout [3,1,2] = 0`
  - `minout [1,5,3,0,2] = 4`
  - `minout [11,5,3,0] = 1`
- How do we compute `minout`?
- The linear time solution via lists involved a rather clever divide and conquer algorithm.
- With arrays the solution is simpler

`minout` via arrays

- Our strategy is the following. Let `m` be the length of the given list ls

`minout` via arrays

- Our strategy is the following. Let `m` be the length of the given list ls
  - Initialize an array with indices `0, .., m` with `0`.

`minout` via arrays

- Our strategy is the following. Let `m` be the length of the given list ls
  - Initialize an array with indices `0, .., m` with `0`.
  - Create an associative list
    `[(i,1) | i <- ls, 0 <= i, i <= m]`

`minout` via arrays

- Our strategy is the following. Let `m` be the length of the given list ls
  - Initialize an array with indices `0, .., m` with `0`.
  - Create an associative list
    `[(i,1) | i <- ls, 0 <= i, i <= m]`
  - Accumulate values from this associative list using the function
    `f x y = y`

`minout` via arrays

- Our strategy is the following. Let `m` be the length of the given list ls
  - Initialize an array with indices `0, .., m` with `0`.
  - Create an associative list
    `[(i,1) | i <- ls, 0 <= i, i <= m]`
  - Accumulate values from this associative list using the function
    `f x y = y`
  - The index of the first entry in the array with `0` is the answer.

minout via arrays ...

```
import Data.Array

minout ls = firstZero 0
  where

  m = length ls
  f x y = y

  myArray = accumArray f 0 (0,m)
      [(i,1) | i <- ls, 0 <= i, i <= m]

  firstZero :: Int -> Int
  firstZero i
   | (myArray!i == 0) = i
   | otherwise = firstZero  (i+1)
```

Two dimensional arrays

- The definition of an array makes no reference to a dimension.

Two dimensional arrays

- The definition of an array makes no reference to a dimension.
- So, two or k-dimensional arrays are essentially same, with just a different sent of indices.

Two dimensional arrays

- The definition of an array makes no reference to a dimension.
- So, two or k-dimensional arrays are essentially same, with just a different sent of indices.
- Here is way to generate an $n \times n$ identity matrix.

```
idMat n = accumArray f 0 ((0,0),(n-1,n-1))
            [((i,i),1) | i <- [0..(n-1)]]
  where
  f x y = y
```