Functional Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

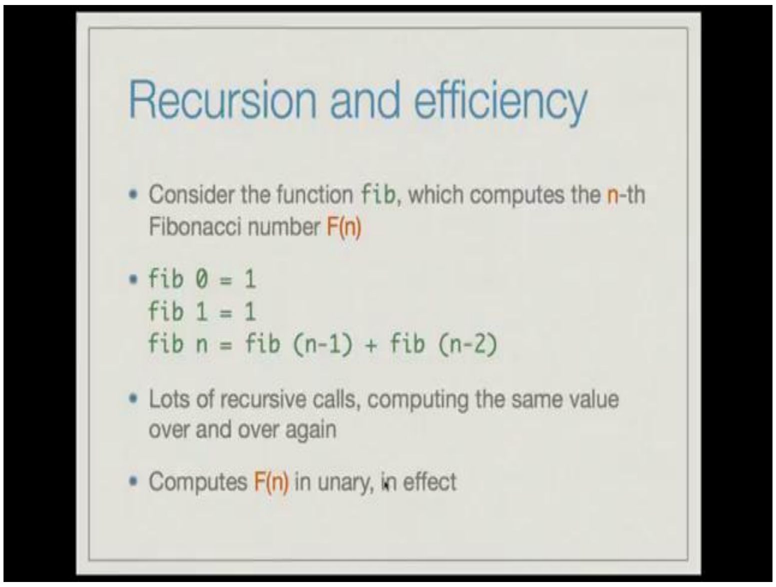
Module # 07

Lecture - 01

Arrays

In this lecture, we shall look at a new topic namely arrays in Haskell.

(Refer Slide Time: 00:07)



Recursion and efficiency

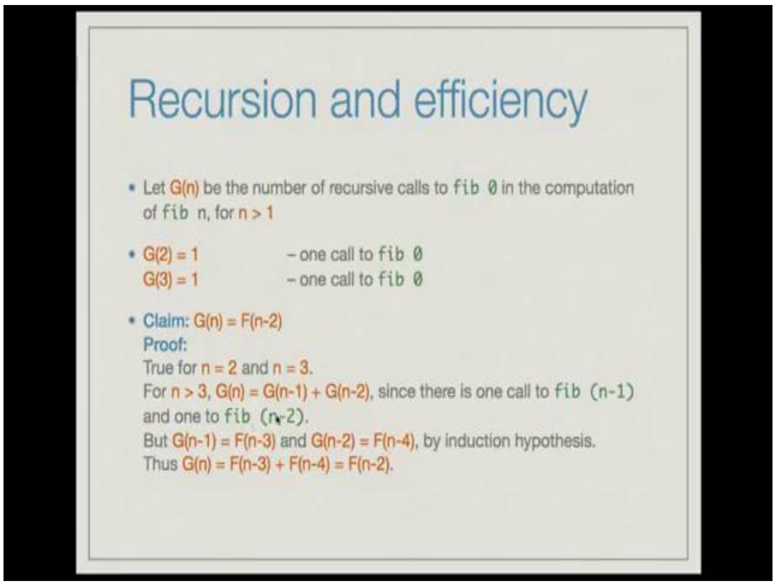
- Consider the function `fib`, which computes the n-th Fibonacci number F(n)

-
  ```
  fib 0 = 1
  fib 1 = 1
  fib n = fib (n-1) + fib (n-2)
  ```
- Lots of recursive calls, computing the same value over and over again
- Computes F(n) in unary, in effect

Arrays are generally used to make programs more efficient, we illustrate this with the following example. Consider the function fib, which computes the nth Fibonacci number F of n, fib 0 equals 1, fib of 1 equals 1, fib of n equals fib of n minus 1 plus fib of n minus 2. You can see that this program is straight forward translation of the mathematical definition of the Fibonacci series.

The program is quite simple, but the problem with that is that there are lots of recursive calls computing the same value over and over again. For instance, fib of 3 makes two calls, one to fib of 2 and another to fib of 1, fib of 2 in turn makes a recursive call to fib of 1 and fib of 0. So, you see that already fib 1 is being called twice, if you consider a call like fib of 5, then you can imagine that there are more calls to fib 1, which in a fib recomputes the same value again and again. In fact, one can see that this program computes F of n in unary in effect.

(Refer Slide Time: 01:27)



Recursion and efficiency

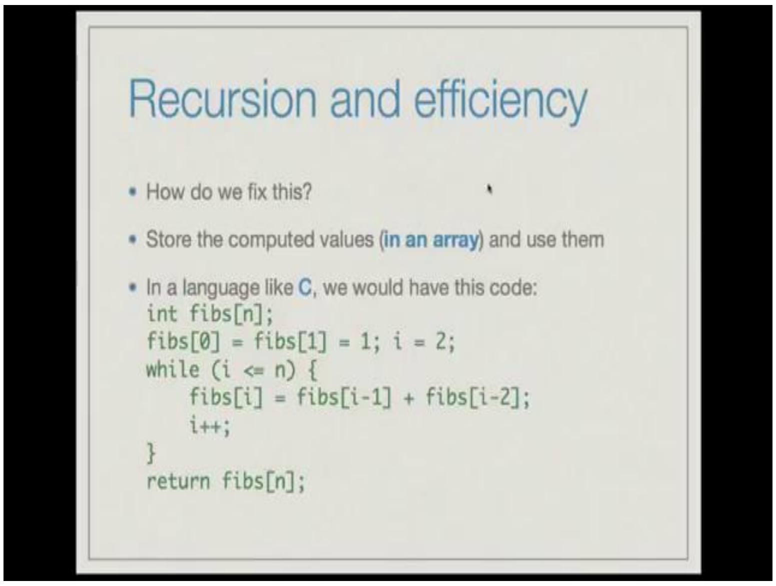
- Let G(n) be the number of recursive calls to `fib 0` in the computation of `fib n`, for n > 1
- G(2) = 1 – one call to `fib 0`
  G(3) = 1 – one call to `fib 0`
- Claim: G(n) = F(n-2)
  Proof:
  True for n = 2 and n = 3.
  For n > 3, G(n) = G(n-1) + G(n-2), since there is one call to `fib (n-1)` and one to `fib (n-2)`.
  But G(n-1) = F(n-3) and G(n-2) = F(n-4), by induction hypothesis.
  Thus G(n) = F(n-3) + F(n-4) = F(n-2).

To make this formal, let G of n be the number of recursive calls to fib 0 in the computation of fib of n for n greater than 1 it is easy to see the G of 2 equals 1, because fib of 2 equals fib of 1 plus fib of 0. So, there is one recursive called fib 0, it is also easy to see that G of 3 equals 1, because fib of 3 equals fib of 2 plus fib of 1, fib of 2 max recursive call to fib 1 and another recursive call it fib 0, fib 1 is defined directly, so there is one recursive call to fib 0.

We claim that G of n equals F of n minus 2, for a proof we see that the statement is true for n equals 2 and n equals 3 for n greater than 3, G of n equals G of n plus n minus 1 plus G of n minus 2. Because there is one call to fib of n minus 1 and one call to fib of n minus 2 in fib of n and there are G of n minus 1 calls to fib 0 in fib n minus 1 and G of n minus 2 calls to fib 0 in fib of n minus 2. But, by induction hypothesis we know that G of n minus 1 equals F of n minus 3 and G of n minus 2 equals F of n minus 4.

Therefore, G of n equals F of n minus 3 plus F of n minus 4, which is F of n minus 2, which is a large number, which grow exponentially in the size of n. Those we see that there are exponential many calls to fib of 0 in a computation of fib of n.

(Refer Slide Time: 03:09)



Recursion and efficiency

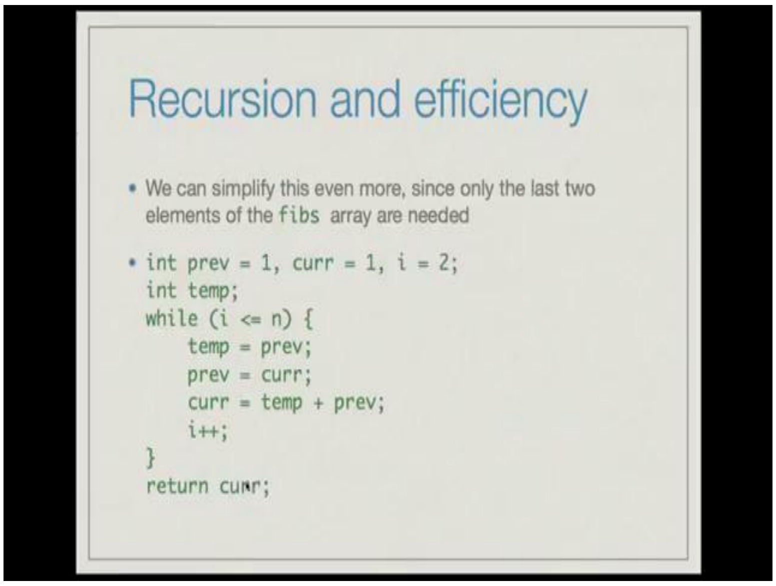
- How do we fix this?
- Store the computed values (**in an array**) and use them
- In a language like **C**, we would have this code:

```
int fibs[n];
fibs[0] = fibs[1] = 1; i = 2;
while (i <= n) {
    fibs[i] = fibs[i-1] + fibs[i-2];
    i++;
}
return fibs[n];
```

How do you fix this situation? One easy way to do this in other language is, is to just store the computed values in an array and use the values from the array rather than re computing them again and again. In a language like C for instance, we would have the following code, we have an array of fibs which store all the Fibonacci numbers from f 0 to f n minus 1. You initialize the array by saying fibs of 0 equals fibs of 1 equals 1, then you fill in entries in the array from index 2 till index n minus 1.

So, you initialize i equals 2, while i is less than n, you just compute fibs of i is to be equal to fibs of i minus 1 plus fibs of i minus 2. This computation does not involve a recursive call, rather it just fix a two values from the array, adds them and stores it in the new value of the array. Finally, you return fibs of n, which is the nth entry in the array; this program actually takes time proportional to n rather than proportional to 2 to the n as in the earlier case.

(Refer Slide Time: 04:40)



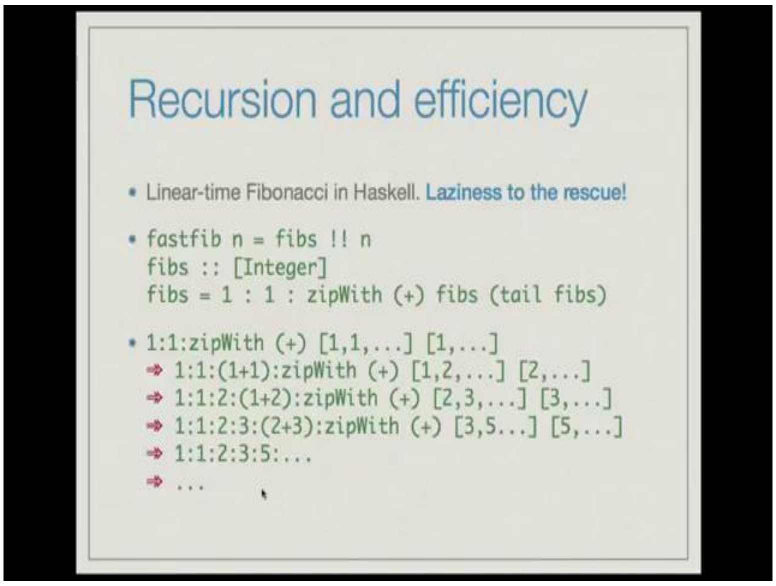
Recursion and efficiency

- We can simplify this even more, since only the last two elements of the `fibs` array are needed

-
  ```
  int prev = 1, curr = 1, i = 2;
  int temp;
  while (i <= n) {
      temp = prev;
      prev = curr;
      curr = temp + prev;
      i++;
  }
  return curr;
  ```

You can simplify this program even more by observing that only the last two elements of the fibs array are ever needed. So, you can have two variables, previous and current storing the last two values of the fibs array, then you run a loop from for i equals 2 to n, where you move the current value to the previous, to the value previous and you move the sum of the two variables to current. In this way you just keep track of the last two entries of the fibs array and finally, you return the value of the current variable.

(Refer Slide Time: 05:26)



Recursion and efficiency

- Linear-time Fibonacci in Haskell. **Laziness to the rescue!**

-
  ```
  fastfib n = fibs !! n
  fibs :: [Integer]
  fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
  ```

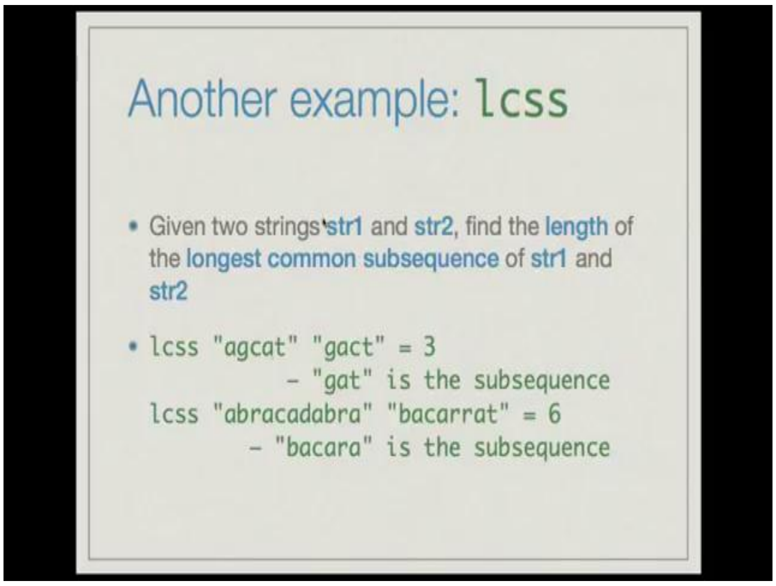
-
  ```
  1:1:zipWith (+) [1,1,...] [1,...]
  ⇒ 1:1:(1+1):zipWith (+) [1,2,...] [2,...]
  ⇒ 1:1:2:(1+2):zipWith (+) [2,3,...] [3,...]
  ⇒ 1:1:2:3:(2+3):zipWith (+) [3,5...] [5,...]
  ⇒ 1:1:2:3:5:...
  ⇒ ...
  ```

We can also program a linear time Fibonacci function in Haskell by using the power of laziness. Here is the function fastfib n, which just builds the Fibonacci series in a list and extracts the nth entry, fibs is a function with signature list of integer and it is given by this program fibs equals 1 colon 1 colon zipWith of plus fibs and tail of fibs. The way the computation unfolds is as follows, this is the expression 1 colon 1 colon zipWith of plus fibs and tail of fibs.

But, fibs we know now is 1 comma 1 comma some other entries, tail of fibs is 1 comma some other entries. So, now, zipWith will add this one with this one and do a zipWith of plus on the tail of the two list. So, it will be the tail of this list, but we now know that the second entry in this list is 2, because that has been computed. So, you will have 1 colon 1 colon 2 colon zipWith using plus of 1 comma 2 comma dot, dot, dot and the tail of this list which is 2 comma dot, dot, dot.

This will turn arise to 1 colon 1 colon 2 colon 3 colon zipWith of plus on the two lists, 2 comma 3 comma dot, dot, dot and it is tail, which is 3 comma dot, dot, dot and so on. You see that you finally, end up with the list 1 colon 1 colon 2 colon 3 colon 5 colon etcetera, which is an infinite list that contains all the Fibonacci numbers extracting the nth element gives you the nth Fibonacci number.

(Refer Slide Time: 07:31)



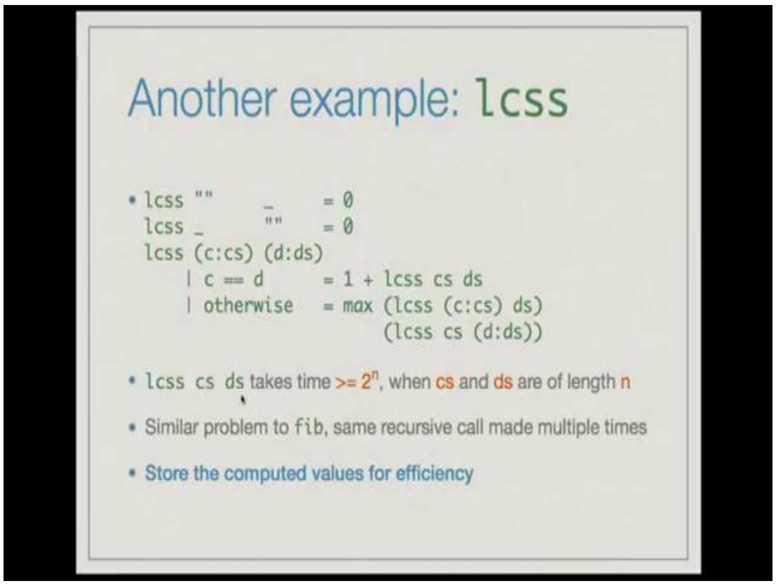
Another example: `lcss`

- Given two strings str1 and str2, find the length of the longest common subsequence of str1 and str2

-
  ```
  lcss "agcat" "gact" = 3
                 - "gat" is the subsequence
  lcss "abracadabra" "bacarrat" = 6
              - "bacara" is the subsequence
  ```

That is fine, but now let us consider the another example; this is the example of computing the longest common sub sequence of two strings, string 1 and string 2. We in fact, want to just compute the length of the longest common sub sequence of string 1 and string 2. For instance lcss agcat and gact equals 3, because gat is a sub sequence of gact and gat is also a sub sequence of gact, that is the longest common sub sequence, lcss of abracadabra and bacarrat equals 6, because bacarrat is a common sub sequence and that is the longest sub sequence b a c a r a, here also you see b a c a r a.

(Refer Slide Time: 08:27)



Another example: lcss

-
  ```
  lcss ""  _   = 0
  lcss _   ""  = 0
  lcss (c:cs) (d:ds)
       | c == d     = 1 + lcss cs ds
       | otherwise  = max (lcss (c:cs) ds)
                          (lcss cs (d:ds))
  ```
- lcss cs ds takes time >= 2^n, when cs and ds are of length n
- Similar problem to fib, same recursive call made multiple times
- Store the computed values for efficiency

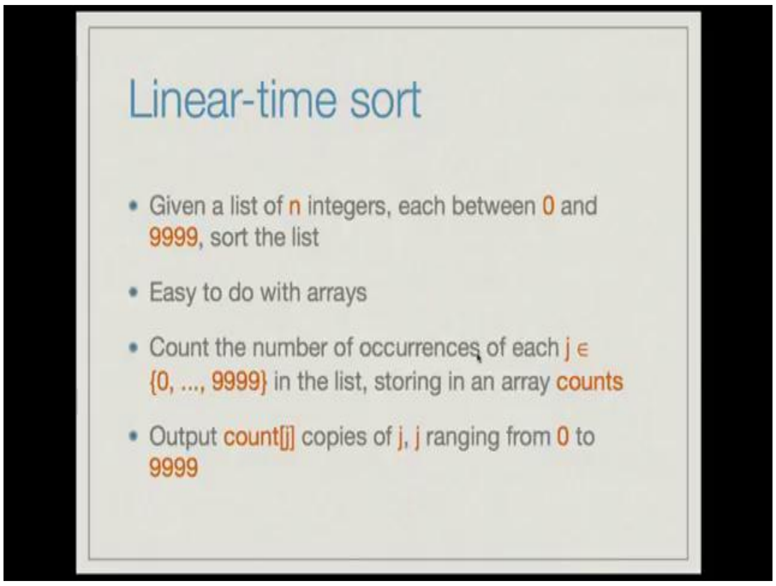
How do you program this? Well, lcss of the empty string with anything is 0, because the empty string does not have any sub sequence. Or you could say the empty string has only itself as a sub sequence and the empty string is always sub sequence of any other sequence, lcss of anything else and the empty string is also 0, lcss of two non empty string, which are given by c colon c is and d colon d is, is computed as follows.

In the case that c is equal to d, then you know that any sub sequence any common sub sequence of c s and d s can be extended by adding c to the front and that will give you a sub sequence of length one longer. So, if you have something to be the longest common sub sequence of c s and d s, which is computed by lcss c s d s we can always add 1 to it, you can always add one letter to the front namely c and get a common sub sequence whose length is 1 longer than the length of lcss, c s and d s.

Otherwise this is the case when c is not equal to d, then it is clear that the first letter is not the same. So, therefore, in the longest common sub sequence of c colon c s and d colon d s is either the longest common sub sequence of c colon c s with d s or the longest common sub sequence of c s with d colon d s which is exactly, what we are doing here. In the case that c is not equal to d, the longest common sub sequence, the length of the longest common sub sequence of these two lists is the same as max of lcss c colon c s d s and lcss c s and d colon d s.

From the proof that lcss of c s and d s, which are two strings takes time at least two to the n, where c s and d s are of length n. There is a similar problem to fib, in that the same recursive call is made multiple times with the same arguments. So, the easiest way how is to store the computed values for efficiency, store the values computed and extract the values later rather than re computing them.

(Refer Slide Time: 10:57)



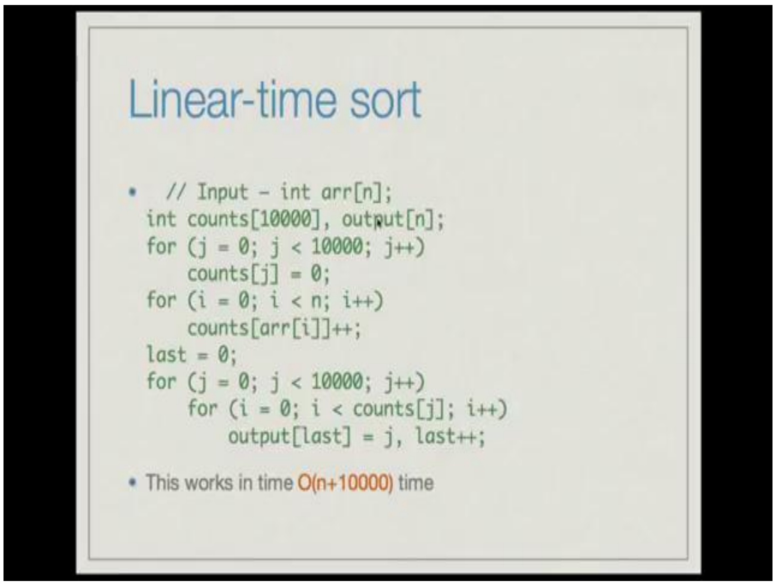
Linear-time sort

- Given a list of n integers, each between 0 and 9999, sort the list
- Easy to do with arrays
- Count the number of occurrences of each j ∈ {0, ..., 9999} in the list, storing in an array counts
- Output count[j] copies of j, j ranging from 0 to 9999

Here is another example, which is that of linear time sort, we already know about that programs like merge sort take order n log n time. And one can even prove a lower bound for sorting that to sort a list, to sort an array of length n you need n log n time, but these are for algorithms which are based on comparison. In linear time sort, we will follow a different idea under a crucial assumption, we have given a list of n integers; such that each integer is between 0 and 9999 sort.

So, so each of the integers lie between a pre specified range and now, we want to sort this list. Suppose the list were stored in arrays, what we could do is to count the number of occurrences of each number between 0 and 9999 in this list and store it in a different array, which stores all the counts. Now, we just need to output count j copies of j, where j ranges from 0 to 9999 that will actually produce the sorted array.

(Refer Slide Time: 12:20)



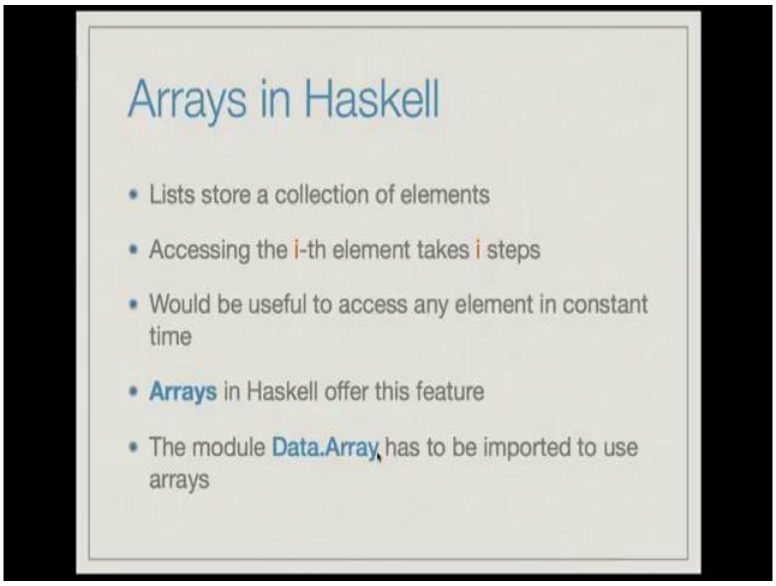
Linear-time sort

-
  ```
  // Input - int arr[n];
  int counts[10000], output[n];
  for (j = 0; j < 10000; j++)
      counts[j] = 0;
  for (i = 0; i < n; i++)
      counts[arr[i]]++;
  last = 0;
  for (j = 0; j < 10000; j++)
      for (i = 0; i < counts[j]; i++)
          output[last] = j, last++;
  ```
- This works in time O(n+10000) time

Here is the program that realizes it, you have an array counts which holds 10000 entries, entries when it from counts 0 to counts 9999. You have an input array which is of size n, you have an output array which is of size n, you first fill in the counts array with the appropriate, you first initialize the counts array to 0. And then, you walk through the input array for i equals 0 till n minus 1, you look at each array entry and then, update the appropriate counts entry.

So, if array i equals 500 let say, counts of 500 will be incremented by 1. Now, the final part is just to go through the counts array and output so many copies of j, for j ranging from 0 to 9999, you look at counts j and output, so many copies of j, this will give you the sorted version of the original array. This algorithm works in time n plus10000, so if n is much larger than 10000, we are actually manage to sort an array of size n in linear time.

(Refer Slide Time: 13:48)

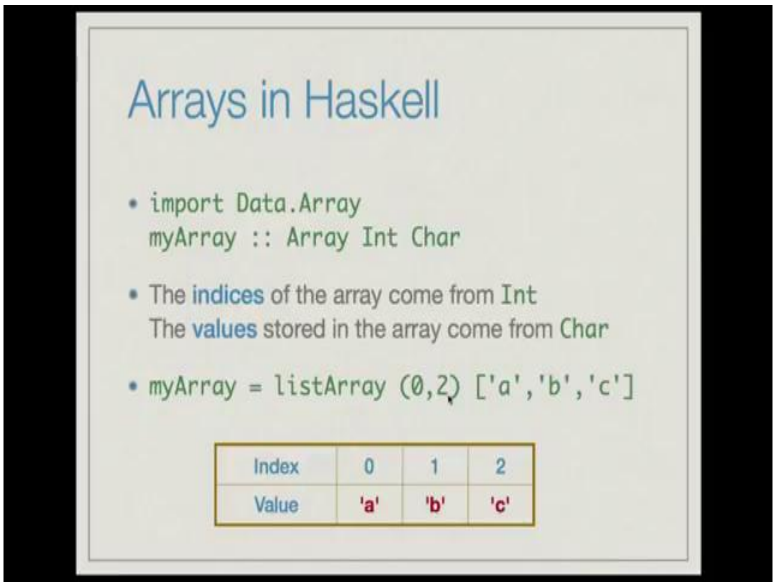


Arrays in Haskell

- Lists store a collection of elements
- Accessing the i-th element takes i steps
- Would be useful to access any element in constant time
- **Arrays** in Haskell offer this feature
- The module **Data.Array** has to be imported to use arrays

To see all this in Haskell, we need to actually use arrays. We know that lists store a collection of elements, the crucial point about lists is that accessing the ith element takes i steps. It would be useful to access any element in constant time and this feature is offered by arrays in Haskell. To use arrays, you need to import the module data dot array.

(Refer Slide Time: 14:23)



Arrays in Haskell

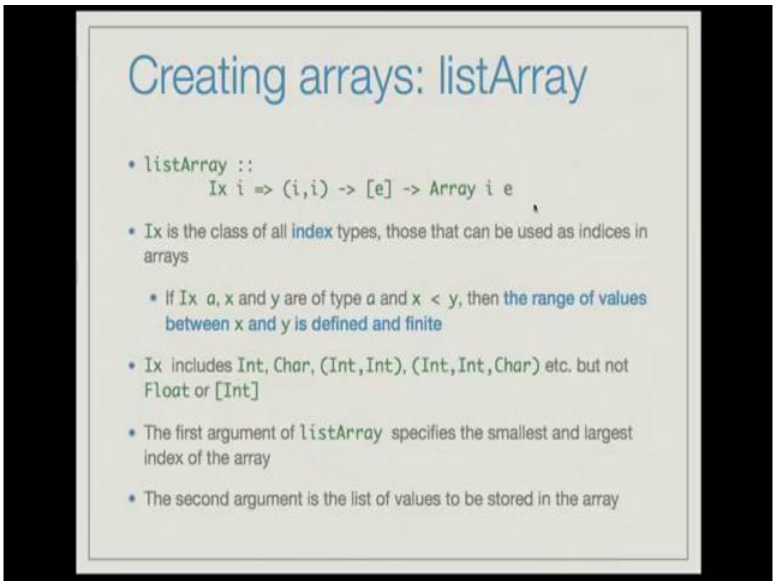
-
  ```
  import Data.Array
  myArray :: Array Int Char
  ```
- The **indices** of the array come from `Int`
  The **values** stored in the array come from `Char`
- `myArray = listArray (0,2) ['a','b','c']`

Index	0	1	2
Value	'a'	'b'	'c'

Here is how we use this import data dot array and then, you want to declare an array, let say myArray, the type of an array is given like this, for instance myArray is of type array

Int char. Here the indices of the array come from Int and the values stored in the array come from char. For instance, let say that myArray equals listArray 0 to a b, the list a b c, then it produces this array. This is an array with three indices 0 1 and 2 and the values a at index 0, b at index 1 and c at index 2. This notation says that we have to create an array from the given list with indices lying between 0 and 2.

(Refer Slide Time: 15:31)



Creating arrays: listArray

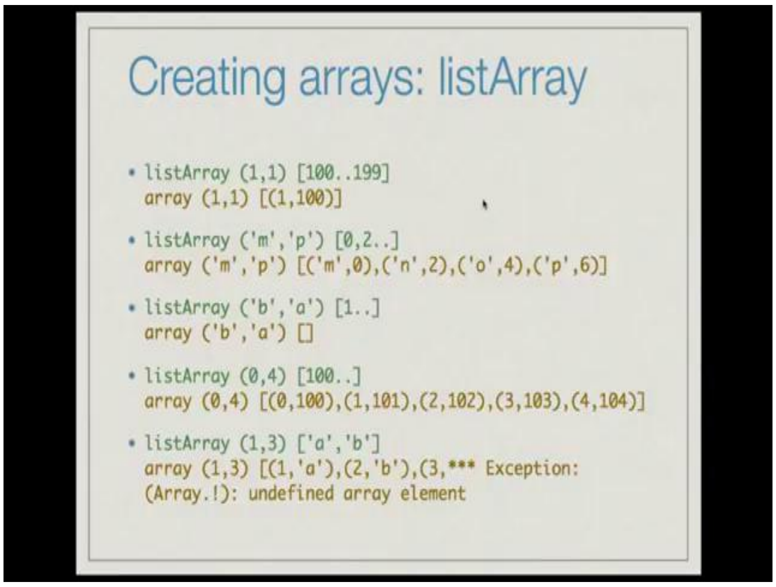
- `listArray :: Ix i => (i,i) -> [e] -> Array i e`
- Ix is the class of all index types, those that can be used as indices in arrays
  - If Ix a, x and y are of type a and x < y, then the range of values between x and y is defined and finite
- Ix includes Int, Char, (Int,Int), (Int,Int,Char) etc. but not Float or [Int]
- The first argument of listArray specifies the smallest and largest index of the array
- The second argument is the list of values to be stored in the array

We look at lisArray and more detail, now listArray has type I x i implies the pair i comma i array list e arrow array e I x is the type class of all index types, which are those type that can be used indices in array. If I x a equals and x and y are of type a and x is less than y, then the range of values between x and y is defined and finite this is the property of the type class I x. For instance I x includes the types Int char pair Int comma Int the triple Int comma Int comma char etc, but not float or list Int.

Because, if you take two floating point values x and y and let, say x is less than y, then the range of values between x and y is not finite. Similarly, if you take the list, let us say the list consisting of the single element 1 and the list consisting of the single element 2 there are infinitely when you list that lie in between these two. The list consisting of 1 comma 2 the list 1 comma 1 comma 2 the list 1 comma 1 comma 1 comma 2 etc. All lie in between the single term list 1 and the single term list 2, therefore, the list cannot be used as an index type.

The first argument of the listArray function specifies the smallest and largest index of the array i comma i the second argument is the list of values to be stored in the array. And finally, the output is an array to index type is i and whose value type is no.

(Refer Slide Time: 17:29)



Creating arrays: listArray

```
• listArray (1,1) [100..199]
  array (1,1) [(1,100)]

• listArray ('m','p') [0,2..]
  array ('m','p') [('m',0),('n',2),('o',4),('p',6)]

• listArray ('b','a') [1..]
  array ('b','a') []

• listArray (0,4) [100..]
  array (0,4) [(0,100),(1,101),(2,102),(3,103),(4,104)]

• listArray (1,3) ['a','b']
  array (1,3) [(1,'a'),(2,'b'),(3,*** Exception:
  (Array.!): undefined array element
```

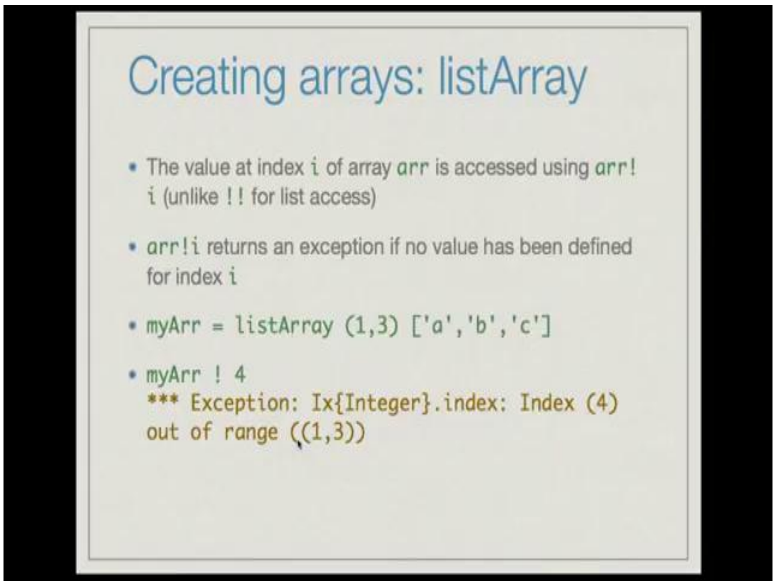
For example, listArray to apply to the pair 1 comma 1 and the list 100 dot, dot, dot 199 will give you the following array, array 1 comma 1, which is the range of the indexes the indexes range is 1 2 1, which means there is only one index. And a list itself the array itself consist of one entry with in index 1 and value 100. The values in the array of full in the order there presented in the list, so therefore 100 is associated with the index 1 form and not say 129.

Here is an another example listArray the pair m comma p the m and p are a characters and, where you want to stored values from 0 comma 2 dot, dot, which is the infinite list of all the even positive integers will give you the following array. Array with the index bounds m and p and the entries of the array there are 4 entries in the array in the index m stories the value 0 index n stores the value two index 0 stores the value 4 index p stores the value 6. Here one another example listArray b comma a and a value is from one dot dot dot, which is the infinite list will give you the empty array.

This is because the, a character a is actually less than the character b. So, therefore, cannot be index the prerequisite of the array to have at least one entry is that the upper bound of the index should be actually greater than or equal to the lower bound of the index. Here is another example listArray 0 comma four 100 dot, dot, this will be array with bounds for the index 0 and 4 and entries is being 0 comma 100, 1 comma 101, 2 comma 102, 3 comma 103, 4 comma 104.

Here is another example listArray 1 comma 3, which tells that there are three indices 1 to and 1 3 and the list itself it is only two elements a comma b this will actually produce an exception. Because Haskell tries to fill a value corresponding to index 3, but it cannot find the any element, so it gives an exception saying undefined array element.

(Refer Slide Time: 20:24)



Creating arrays: listArray

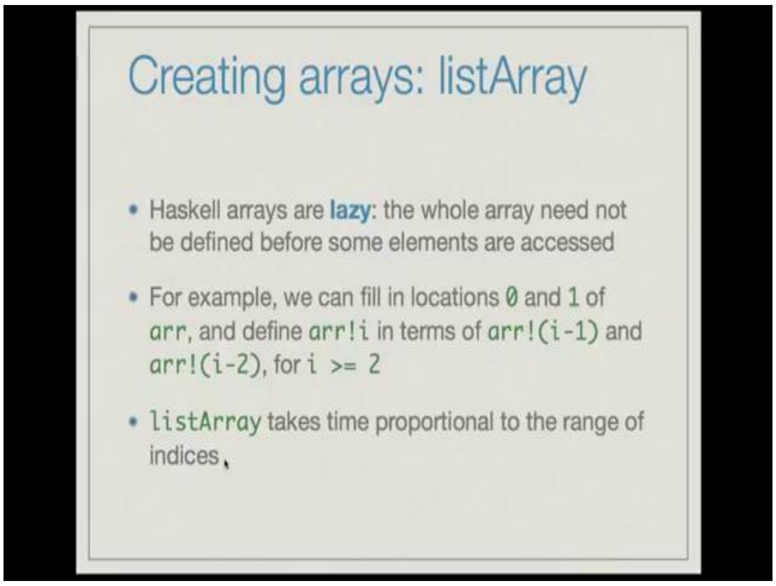
- The value at index `i` of array `arr` is accessed using `arr!i` (unlike `!!` for list access)
- `arr!i` returns an exception if no value has been defined for index `i`
- `myArr = listArray (1,3) ['a','b','c']`
- `myArr ! 4`

```
*** Exception: Ix{Integer}.index: Index (4)
out of range ((1,3))
```

The value at index i of n array is access using the single exclamation mark like, so array exclamation mark i unlike the double exclamation mark for list access. So, a r r exclamation i returns the ith value in the array, but it returns an exception is no value has been define for index i which is why in the earlier slide, we saw that we got an exception for creating an array with three index value, but one with only two values.

Suppose, we created an array, myArray using listArray 1 comma 3, a comma b comma c, now if you try to access fourth element in the array, myArray exclamation mark 4 will again get exception saying the index 4 is out of range.

(Refer Slide Time: 21:25)

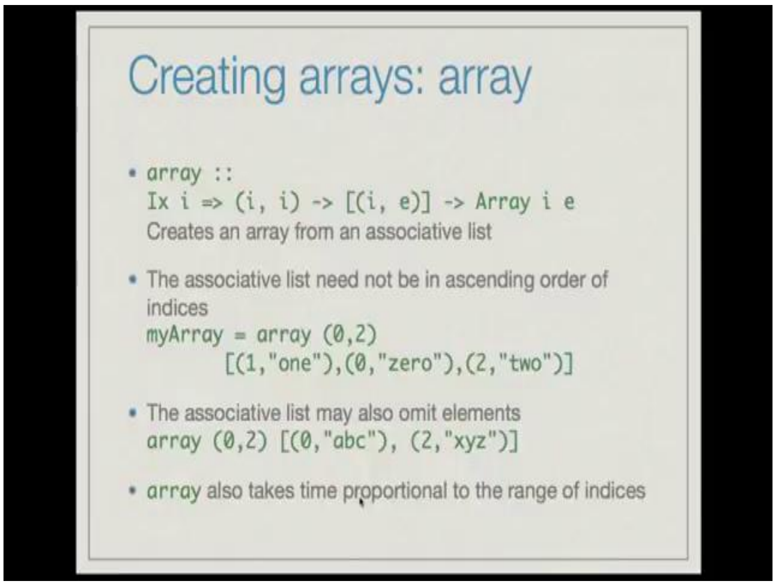


Creating arrays: listArray

- Haskell arrays are **lazy**: the whole array need not be defined before some elements are accessed
- For example, we can fill in locations `0` and `1` of `arr`, and define `arr!i` in terms of `arr!(i-1)` and `arr!(i-2)`, for `i >= 2`
- `listArray` takes time proportional to the range of indices

An important point to knot is that Haskell arrays are lazy; the whole array need not be defined before some elements are accessed. For example, we can fill in location 0 and 1 of the array and define the ith element of the array in terms of the i minus first element and i minus second element. The exactly remains cent of the array version of the computing Fibonacci series in c, another point to note is that listArray takes time propositional to range of the indices. So, if there are k indices the call to listArray takes time order k.

(Refer Slide Time: 22:15)



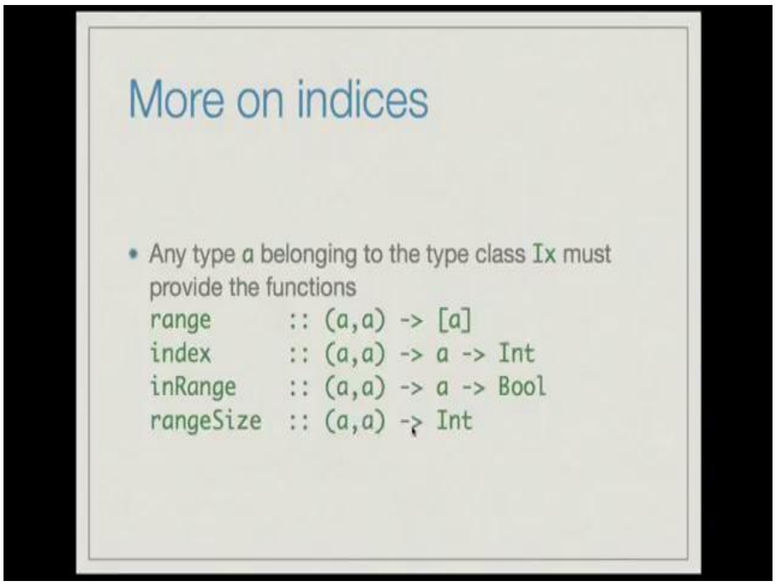
Creating arrays: array

-
  ```
  array ::
  Ix i => (i, i) -> [(i, e)] -> Array i e
  ```
  Creates an array from an associative list
- The associative list need not be in ascending order of indices
  ```
  myArray = array (0,2)
          [(1,"one"),(0,"zero"),(2,"two")]
  ```
- The associative list may also omit elements
  ```
  array (0,2) [(0,"abc"), (2,"xyz")]
  ```
- `array` also takes time proportional to the range of indices

There is another way to create arrays, which is to use the call array, which is use to function array. So, to type is as follows I x i implies the pair i comma i arrow list of pair is i comma e arrow array i e. So, instead of produce the listArray only values, which was a list of elements of type e, we are now providing associative list and creating an array out of this.

They associative list need not be in ascending order of the indices for instance you could create an array as follows myArray equals array the pair 0 comma 2 and the entries or 1 comma the string 1 0 comma the string z e r o 2 comma the string t w o. But, notices that the elements of the associative list are presented in not present in ascending order the indices. They associative list may also omit element, so you have array 0 comma 2 0 abc comma 2 XYZ there is no entry for 1, this call also takes time propositional to the range of the indices.

(Refer Slide Time: 23:37)



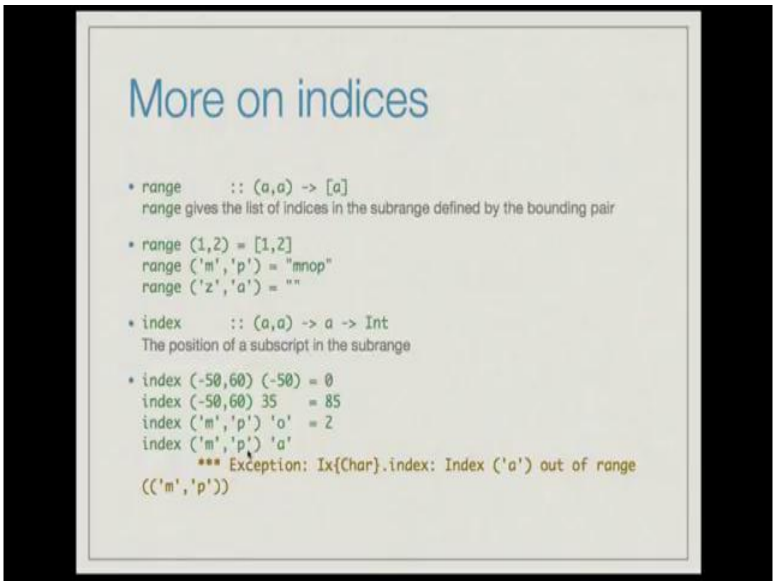
More on indices

- Any type a belonging to the type class Ix must provide the functions

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

Let us look at indices little more detail any type a belonging to the type class I x must provide the following functions. A range, which is function from the pair is the form a comma a list a index, which is a function, from which is a function signature pair a comma a arrow a arrow Int. In range, which as signature pair a comma a arrow a arrow Bool the range size, which as signature a comma a arrow Int.

(Refer Slide Time: 24:12)



More on indices

- range :: (a,a) -> [a]
  range gives the list of indices in the subrange defined by the bounding pair

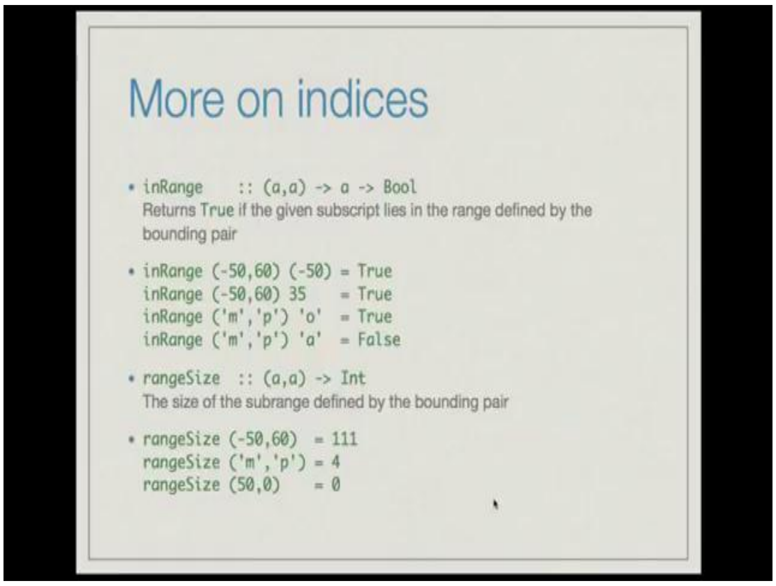
-
  ```
  range (1,2) = [1,2]
  range ('m','p') = "mnop"
  range ('z','a') = ""
  ```
- index :: (a,a) -> a -> Int
  The position of a subscript in the subrange

-
  ```
  index (-50,60) (-50) = 0
  index (-50,60) 35    = 85
  index ('m','p') 'o'  = 2
  index ('m','p') 'a'
          *** Exception: Ix{Char}.index: Index ('a') out of range
  (('m','p'))
  ```

The range function use the list of indices in the sub range defined by the bounding pair the first a lower bound and the second is upper bound and this list a use all the elements the lie between the first number the first element here and the second element here. For instance range 1 comma 2 is the list 1 comma 2 range of character m comma character p is the string m n o p range of character z comma character a is the empty string, because a is smaller than z.

Index given position of the subscript in the range for instance index of minus 50 in the sub range define by minus 50 comma 60 is 0, index of 35 in the sub range define by minus 50 and 60 is 85. Because, you go from fifty to 0 and then from 1 to 35, index of o the character o in the range define by the character m and character p this 2, because you go m is 0 n is 1 and the o is 2. Index of a, the character a in the range define by m comma p will give a exception, because a is out of the range m comma p.

(Refer Slide Time: 25:38)



More on indices

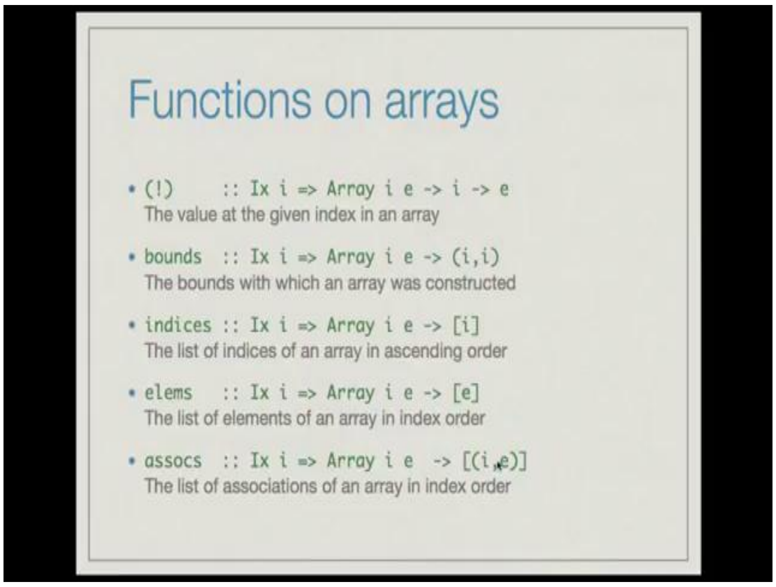
- `inRange    :: (a,a) -> a -> Bool`
  Returns True if the given subscript lies in the range defined by the bounding pair

-
  ```
  inRange (-50,60) (-50) = True
  inRange (-50,60) 35    = True
  inRange ('m','p') 'o'  = True
  inRange ('m','p') 'a'  = False
  ```
- `rangeSize  :: (a,a) -> Int`
  The size of the subrange defined by the bounding pair

-
  ```
  rangeSize (-50,60)  = 111
  rangeSize ('m','p') = 4
  rangeSize (50,0)    = 0
  ```

In range is the function that checks whether a given element lies one the range are for instance in range of minus 50 comma 60 applied on minus 50 will be true. You can also check that o is inside the range define by m and p, but a is not inside the range define and by m and p, so in range return false. Range size just give the size of the range define by the lower bound and the upper bound for instance range size apply to minus 50 comma 60 will give you 111. Range size apply to the character m comma the character p will give you 4 range size defined by 50 comma 0 will give you 0, because 0 was strictly less than 50.

(Refer Slide Time: 26:29)



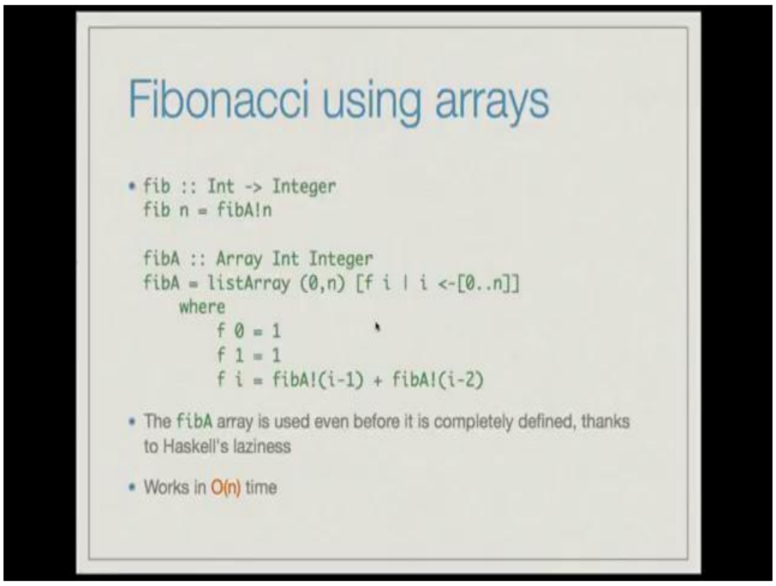
Functions on arrays

- `(!)      :: Ix i => Array i e -> i -> e`
  The value at the given index in an array
- `bounds   :: Ix i => Array i e -> (i,i)`
  The bounds with which an array was constructed
- `indices  :: Ix i => Array i e -> [i]`
  The list of indices of an array in ascending order
- `elems    :: Ix i => Array i e -> [e]`
  The list of elements of an array in index order
- `assocs   :: Ix i => Array i e  -> [(i,e)]`
  The list of associations of an array in index order

Here are some more function of an arrays the exclamation mark we mention earlier is to access an array entry it as signature I x i implies array i e arrow i arrow e, so gives the value at given induction an array. Bounds is an function that takes array i e is an argument and returns the lower and upper bounds are of the indices. The bounds which is the array was originally constricted. Indices a function that the list indices of a array as an ascending order rather than providing this is the pair it produces the list of all indices.

Elems is the function that provides produces the list of all elements of an array index order. Assoc is a function that list all association of array in order of in the ascending order of index, if the signature is I x i implies array i e arrow list of pairs i comma e.

(Refer Slide Time: 27:42)



Fibonacci using arrays

```
• fib :: Int -> Integer
  fib n = fibA!n

  fibA :: Array Int Integer
  fibA = listArray (0,n) [f i | i <-[0..n]]
      where
          f 0 = 1
          f 1 = 1
          f i = fibA!(i-1) + fibA!(i-2)
```

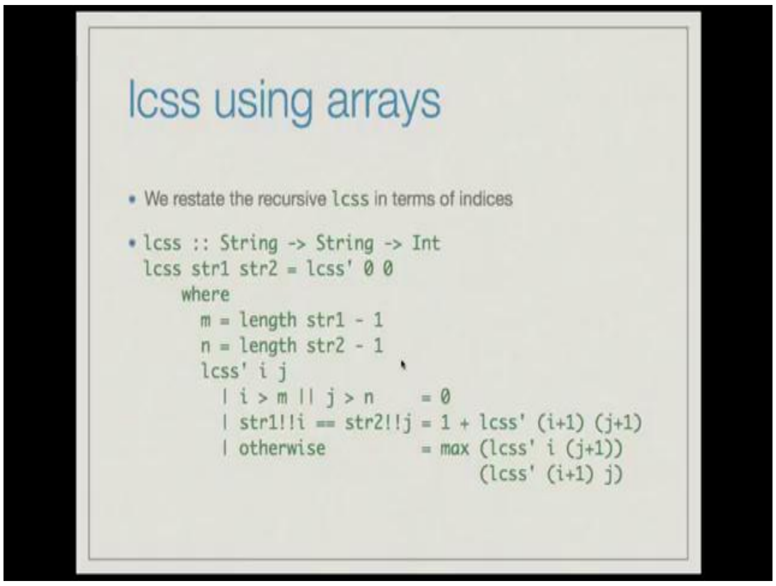
- The fibA array is used even before it is completely defined, thanks to Haskell's laziness
- Works in O(n) time

Let us now, get back to original example that of computing Fibonacci numbers, but now using arrows. So, is a function the signature Int arrow integer fib of n equals fib a exclamation mark n, which is access in the nth element of with the array fib a fib a is array Int integer the indices are from Int in the values are integer. And now, use the fact that Haskell arrays are lazy fib a is define to be listArray 0 comma n, where the ith entry is given by the function f f i.

You create an array out of the list consisting of all values of f i f i for i range from 0 to n f is defined as follows f of 0 plus 1 f of 1 is 1 f of i is not f of i minus 1 plus f of i minus 2, as we would have done in a recursive program. But, fib a exclamation mark i minus 1 plus fib a exclamation mark i minus 2, we are just refer in to the array and picking the elements from the array the i minus first element and the i minus second element. In case the value at fib has not be define it Haskell will easily make a called to f i minus 1.

But, a first time it will make a call to f it will fill the entry in the array, but the second time the called to fib f i minus 1 is made it will just pick out the entry from the array. The fib a array is even before complete redefine thanks to Haskell's laziness and a this program works in order n time, because all its needs to. So, just fill n indices by referring to previous entries in the array.

(Refer Slide Time: 29:43)



lcss using arrays

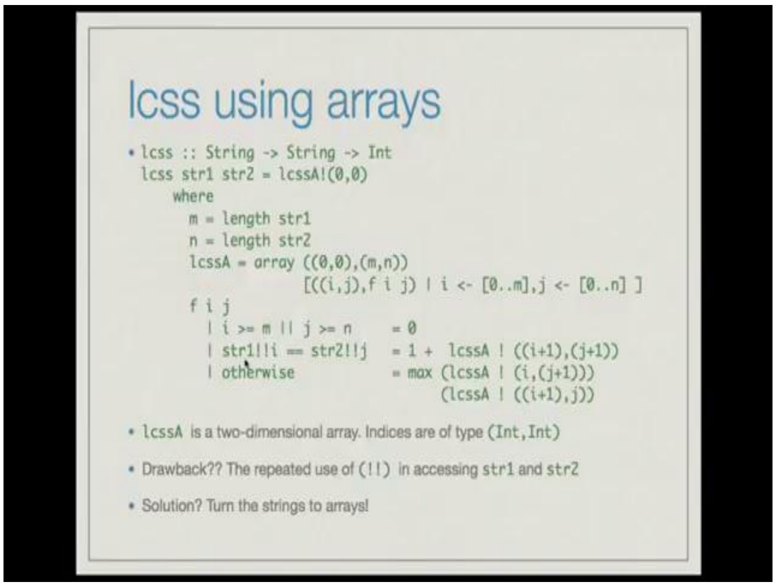
- We restate the recursive `lcss` in terms of indices
-
```
lcss :: String -> String -> Int
lcss str1 str2 = lcss' 0 0
    where
      m = length str1 - 1
      n = length str2 - 1
      lcss' i j
        | i > m || j > n     = 0
        | str1!!i == str2!!j = 1 + lcss' (i+1) (j+1)
        | otherwise          = max (lcss' i (j+1))
                                   (lcss' (i+1) j)
```

Here a show the compute lcss using arrays, we restate the first restore the recursive lcss in the terms of indices lcss is the function with signature string arrow string arrow Int. But, we will work with the version lcss is prime, which works as Int arrow Int arrow Int lcss string 1 and string 2 equals lcss prime 0 0, where lcss prime computes the length of the longest common sub sequences of drop i string 1 drop j i string.

And since, drop 0 string 1 is equal to string 1 and drop string 2 is equal to string 1 equal to string 2 computing lcss prime of 0 comma 0 achieves, what you want to achieve, which is computed lcss of string 1 and string 2. We done be the length of string 1 minus 1 and n be the string length of minus 1 length of string 2 minus 1 these are the last indices, if you will of string 1 and string 2. Now, lcss prime of ij is defined as orders defined as for as if i is greater than m r j is greater than n the value is 0.

Otherwise; the string if the character at the ith location of string 1 is same as character at the jth location of string 2 the result is 1 plus lcss prime of i plus 1 j plus 1. If, the ith character of string 1 is not equal to the jth character string 2, then as earlier we compute the result is in max of lcss prime of i and j plus 1 and lcss prime of i plus 1 and j. Now, we restated the original lcss function in terms of recursion or indices, this we will try to transcribe directly into an array base program.

(Refer Slide Time: 32:07)



lcss using arrays

-
  ```
  lcss :: String -> String -> Int
  lcss str1 str2 = lcssA!(0,0)
      where
        m = length str1
        n = length str2
        lcssA = array ((0,0),(m,n))
                     [((i,j),f i j) | i <- [0..m],j <- [0..n] ]
        f i j
          | i >= m || j >= n      = 0
          | str1!!i == str2!!j    = 1 +  lcssA ! ((i+1),(j+1))
          | otherwise             = max (lcssA ! (i,(j+1)))
                                        (lcssA ! ((i+1),j))
  ```
- lcssA is a two-dimensional array. Indices are of type (Int,Int)
- Drawback?? The repeated use of (!!) in accessing str1 and str2
- Solution? Turn the strings to arrays!

Here is lcss using arrays lcss of string 1 string 2 is the 0 0'th entry of the arrays lcssA the array lcss array a is a two dimensional array whose indices range from 0 comma 0 to n comma n. The entry of the array get i comma j is suppose to be the longest common sub sequence of string 1 starting from index i and string 2 starting from index j or in other words drop i of string 1 and drop j as string 2. In creating lcssA we use the array function to rather than raster array function, because it is easier to provide the values of the array in terms of an associative list.

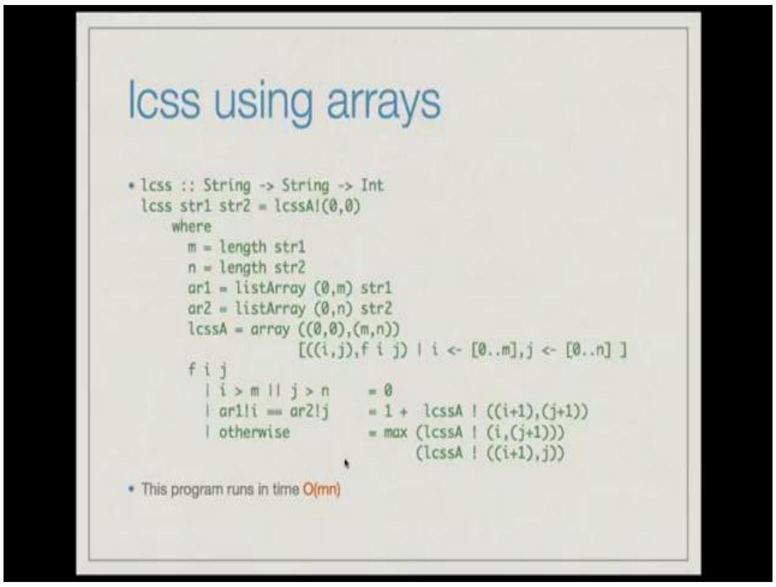
So, we define lcssA as array with range 0 comma 0 to n comma n and entry is of the form the part i comma j comma the value f applied to inj. So, this is the index and this is a value and the index and value is givens a pair, where i ranges from 0 to n and j ranges from 0 to n. In a two dimensional array of this form the indices are ordered as follows 0 comma 0 followed by 0 comma 1 followed by 0 comma 2 all the way to 0 comma n, then counts 1 comma 0 1 comma 1 1 comma 2 etc, all the way to 1 comma n, then 2 comma 0 2 comma 1 to 2 comma 2 etc.

Now, f is defined as follows f on i and j is very similar to the index based recursive function that is described earlier if i is greater than or equal to n r if j is greater than or equal to n the values 0. Otherwise; if check is the ith element of string 1 is same as the jth element and a string 2.

In that case the result is 1 plus lcssA exclamation marks i plus 1 comma j plus 1. Notice, that we are instead of making that the recursive called f we just refer to the corresponding element in the array at the appropriate index. Otherwise; if string 1 exclamation, exclamation i is not equal to the jth element of string 2, we define f of ij to be max of lcssA exclamation mark i comma j plus 1 lcssA exclamation mark i plus 1 comma j this is the function.

One minor drawback is that we repeatedly use the exclamation, exclamation in access in string 1 and string 2 here. Recall that result that accessing an element of an array can be done in constant time there is access in ith element of a list takes time order of order I, the solution to be turn the strings themselves into arrays, which is done as follows.

(Refer Slide Time: 35:32)



lcss using arrays

```
• lcss :: String -> String -> Int
  lcss str1 str2 = lcssA!(0,0)
      where
        m = length str1
        n = length str2
        ar1 = listArray (0,m) str1
        ar2 = listArray (0,n) str2
        lcssA = array ((0,0),(m,n))
                    [((i,j),f i j) | i <- [0..m],j <- [0..n] ]
        f i j
          | i > m || j > n     = 0
          | ar1!i == ar2!j     = 1 +  lcssA ! ((i+1),(j+1))
          | otherwise          = max (lcssA ! (i,(j+1)))
                                     (lcssA ! ((i+1),j))
```

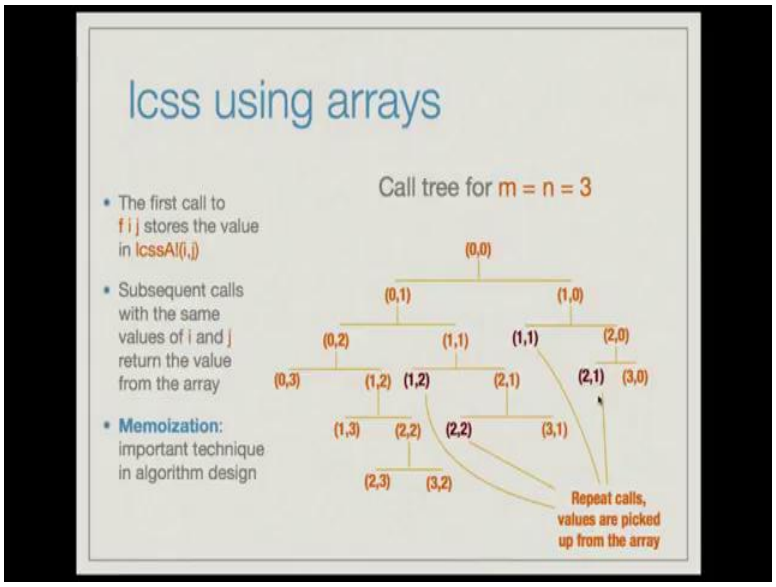
• This program runs in time O(mn)

Here is the version, which works on array rather than strings lcss of string 1 and string 2 is lcss A exclamation 0 comma 0. Where instead of using string 1 and string 2 in the description of a r 1 and a r 2, which are 2 arrays a r 1 is got by listArray 0 comma 1 string 1, where m is the length of string 1 a r 2 is list of 0 comma n string 2, where n is the length of string 2. LcssA is an, it is depend as usual array with indices range from 0 to n comma n and entries of the form i comma j comma f applied to i n j.

Where i ranges from apply to i n j, where i ranges from 0 to n and ranges from 0 to 1 f is now defined in terms of a r 1 and a r 2 f of i j equals 0 i is greater than n j is greater than n. Otherwise; if a r one at position i is equal to ar 2 position j, then you define it to be 1 plus lcss A at position i plus 1 comma j plus 1. Otherwise; you define it as usual the difference is that we access the array rather than the string repeatedly, this program one can check runs at time order n minus 1.

Because, all way to do is filling elements in the array and the number of elements in the array the number of indices of this array is n plus 1 times n plus 1. So, the program runs in time order m one which is a last improvement over to the n that we had earlier.

(Refer Slide Time: 37:24)



lcss using arrays

- The first call to f i j stores the value in lcssA!(i,j)
- Subsequent calls with the same values of i and j return the value from the array
- **Memoization:** important technique in algorithm design

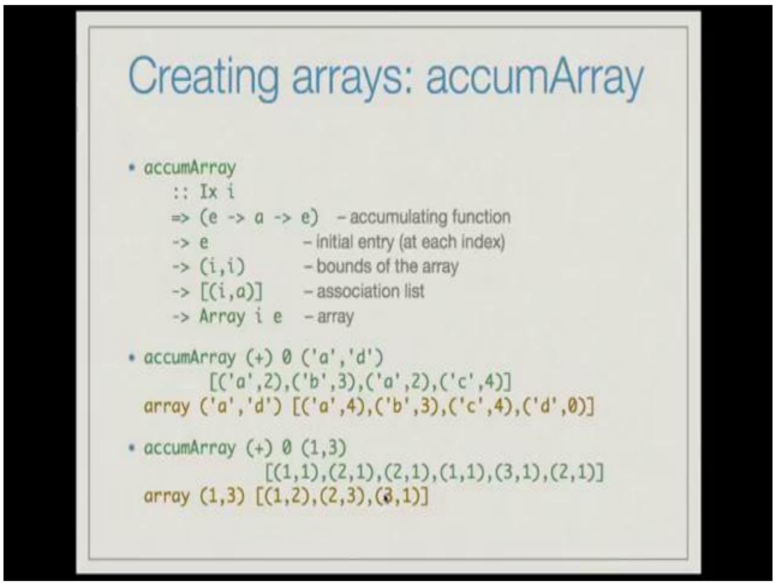
Let us look at the competition in the little more detail, let us look at the call tree for the case when m equals n equals 3 to call tree for l c s s. So, the first called f i j is we are considering f of 3 3 the first called off f of i j stores the value in the array and subsequent calls with the same values y and j. Return the value from the array rather than making the recursive call and this take call me memorization it is an important technique in algorithm design.

Where you translate a recursive algorithm to more efficient version by this storing the values and refer them into later. So, initially there is a call made to f of 0 comma 0 this points calls to f of 0 comma 1 and f of 1 comma 0 possibly it will all it might also span a call to f of1 comma 1, but that occurs here anywhere. F of 0 comma 1 spans call to f of 0 comma 2 and 1 comma 1 0 comma 2 spans calls to 0 comma 3 1 comma 2 this intern leads to calls on 0 comma 3 terminates, because 3, if you recall is greater than or equal to 3 is greater than or equal to n, which is 3.

Therefore, this call terminates by doing the value 0 and that will be stored in the array 1 comma 2 give raise to call to 1 comma 2 and 3 comma 2, 2 comma 2 makes call to 2 comma and 3 comma 2 these two call terminates, because 3 is greater than n and here 3 is greater than n. Now, once this returns we allow to process calls to 1 comma 1 1 comma 1 gives arise to call this to 1 comma 2, 2 comma 1, but 1 comma 2 here, is a repeat call. So, the values are picked up from the array rather than reading leading to a recursive call.

So, there one way this whole tree and 1 comma 2 will not be repeated 1 comma 1 also raise to call 2 comma 1, which will turn gives raise to calls 2 comma 2 and 3 comma 1, but 2 comma 2 is a repeat call earlier here already. So, its values respect from the array and this tree is not repeated under tree 3 comma want terminates, because 3 is greater than or equal to n and so on, you see that there two more repeat calls here. So, in this manner we see that we never have to spend time more than order n and this example also illustrates, how exactly lazy arrays work.

(Refer Slide Time: 40:13)



Creating arrays: accumArray

```
• accumArray
    :: Ix i
    => (e -> a -> e)  – accumulating function
    -> e              – initial entry (at each index)
    -> (i,i)          – bounds of the array
    -> [(i,a)]        – association list
    -> Array i e      – array

• accumArray (+) 0 ('a','d')
        [('a',2),('b',3),('a',2),('c',4)]
  array ('a','d') [('a',4),('b',3),('c',4),('d',0)]

• accumArray (+) 0 (1,3)
        [(1,1),(2,1),(2,1),(1,1),(3,1),(2,1)]
  array (1,3) [(1,2),(2,3),(3,1)]
```

Here is another way to create an array, which is to use the function called accumArray, accumArray accumulates value into array positions and it works as follows. I is signature is ix i implies e arrow to e arrow e, which is accumulate this function this is similar to the kinds of function you provide as arguments to fold r fold l. In particular this is the kind of function that you would provide as an argument to fold l, e which is an initial entry that will be clear in placed in index of the array i comma i this provides the bounds of the array.

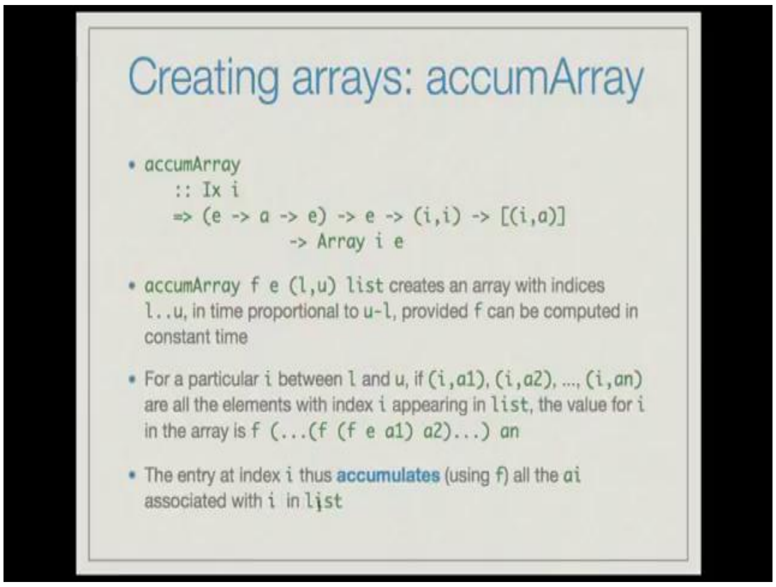
Under association list which is a list of pair of icon away with all there is in produces an array array i e. Instead of how works accumArray with function plus and initial value 0 with indices ranging from the character a to the character b and with the elements coming from the associated list given here, a comma 2 b comma 3 a comma 2 c comma 4 produces the following array. An array with indices ranging from a to d and entries a comma four b comma 3 c comma 4 and d comma 0.

How do you explain the entries? Well 4 is 2 plus 2, 3 is a 3 the 4 here, is the 4 and the 0 here, is the initial value that is placed at the index d. So, you see that accumArray does the following it places the initial value at all entries and then, whenever it encounters of particular index it adds the corresponding value it adds the value that is encountered to the corresponding entry in the array. In this case it adds, because the function that is provides as plus, if it is star it will multiplied.

So, you see that using this function you accumulate all the values associated with this particular index in the list into the array at the appropriate index. True that another example accumArray with function plus and initial value 0 and the bound 1 comma 3 under the list provided by 1 comma 1 2 comma 1 etcetera produces the following array, array bounds 1 comma 3 and entries being 1 comma 2 2 comma 3 and 3 comma 1.

Here we this counting the number of repetitions of each element one occurs twice, here and here and we keep track of the count by initializing the values 0 at array entry 1. And then, using the function post to add 1 every time, we are encounter 1. Similarly, whenever we encounter 2 we add 1 2 the accumulator since 3 2 occurs thrice here, here and here and the final value associated to index 2 in the array is 3 3 occurs ones, so that finally, will have one of the value are index 3.

(Refer Slide Time: 43:50)



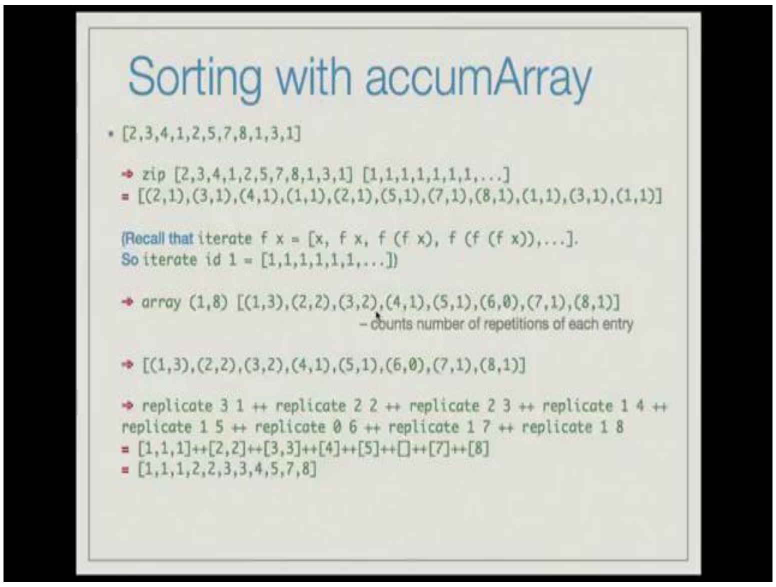
Creating arrays: accumArray

- accumArray
```
    :: Ix i
    => (e -> a -> e) -> e -> (i,i) -> [(i,a)]
              -> Array i e
```
- `accumArray f e (l,u) list` creates an array with indices `l..u`, in time proportional to `u-l`, provided `f` can be computed in constant time
- For a particular `i` between `l` and `u`, if `(i,a1)`, `(i,a2)`, ..., `(i,an)` are all the elements with index `i` appearing in `list`, the value for `i` in the array is `f (...(f (f e a1) a2)...) an`
- The entry at index `i` thus **accumulates** (using f) all the `ai` associated with `i` in `list`

AccumArray f e l comma u list f is the function e is the initial entry l comma u is the lower bound in on the indices and upper bound on the indices on the list is the associated list creates an array with l dot dot u in time proportional to u minus l which is important provided f can be computed in constant time. So, for a particular i between l and u, if i comma a 1 i comma a 2 etcetera i comma a n are all the elements with index i appearing in this list.

The value for i in the array is f apply to e and a 1 the result applied f apply to the result and a 2 f apply to that result and a 3 etcetera and finally, f apply to that result and a n. So, it is a fold l version applied with all the values corresponding to i that occurs in the association list. So, the entry at index i thus accumulates using the function f all the a i values that are associated with i in the list.

(Refer Slide Time: 45:10)



Sorting with accumArray

- [2,3,4,1,2,5,7,8,1,3,1]

```
→ zip [2,3,4,1,2,5,7,8,1,3,1] [1,1,1,1,1,1,1,...]
= [(2,1),(3,1),(4,1),(1,1),(2,1),(5,1),(7,1),(8,1),(1,1),(3,1),(1,1)]

(Recall that iterate f x = [x, f x, f (f x), f (f (f x)),...].
So iterate id 1 = [1,1,1,1,1,1,...])

→ array (1,8) [(1,3),(2,2),(3,2),(4,1),(5,1),(6,0),(7,1),(8,1)]
                                   – counts number of repetitions of each entry

→ [(1,3),(2,2),(3,2),(4,1),(5,1),(6,0),(7,1),(8,1)]

→ replicate 3 1 ++ replicate 2 2 ++ replicate 2 3 ++ replicate 1 4 ++
replicate 1 5 ++ replicate 0 6 ++ replicate 1 7 ++ replicate 1 8
= [1,1,1]++[2,2]++[3,3]++[4]++[5]++[]++[7]++[8]
= [1,1,1,2,2,3,3,4,5,7,8]
```

We can use this to sort and linear time as follows, suppose we are given the list 2 comma 3 comma 4 comma 1 comma 2 comma 5 comma 7, 81, 3, 1. We first create another list an association list as follows by zipping it with one repeated infinitely often that will produce a list of pairs were the first element of inspire on original list and the second element is always 1.

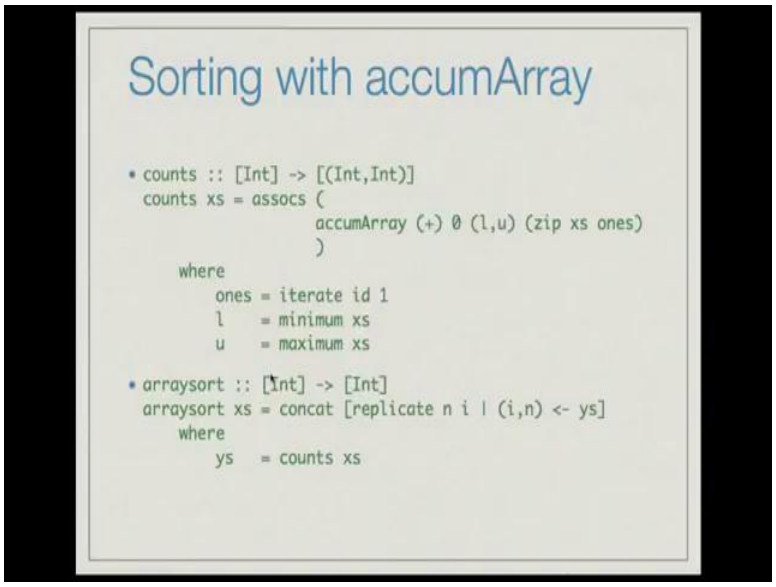
How do you produce this list 1 comma 1 comma 1 comma 1 recall that iterate is a function, which behaves as follows iterate f x equals the list x comma f of x comma f of f of x comma f of f of f of x etcetera. So, you can start with x being 1 and f being the identity function and will get you see that iterate id 1 equals the infinite list consisting only of ones. From this association list will produce on array, array 1 8 the 1 here is the minimum value on the original list and 8 is the maximum value on the original list.

So, we have the bounds of the array and you are produced, this array this is an accumArray therefore, this array stores the count of the number of repetitions of each entry. From this array you going to the list 1 comma 3, 2 comma 2, 3 comma 2 etcetera, you can obtain this list from the array by using the function assoc. Once you have this list, you replicate one thrice and that is done using the function replicate, replicate n i give, repe gives you the list consisting of the value i repeated n times.

So, replicate 3 1 gives of the list 1 comma 1 comma 1 when you replicate 2 twice replicate 2 comma 2 replicate 2 comma replicate of 2 3 that is, because of onto replicate the value 3 twice replicate 1 4 plus plus replicate 1 5 etcetera. Battle produce this list 1 comma 1 comma 1 plus plus 2 comma 2 plus plus 3 comma 3 etcetera. and this will finally, produce this list which is the sorted version of the original list.

AccumArray works in time proportional to the length of the association list and assoc also works on time proportional to the length of the list produced. Therefore, this algorithm works in linear time or order n plus knocks minus min of the elements in the array, if you prefer of the elements in the list.

(Refer Slide Time: 48:05)



Sorting with accumArray

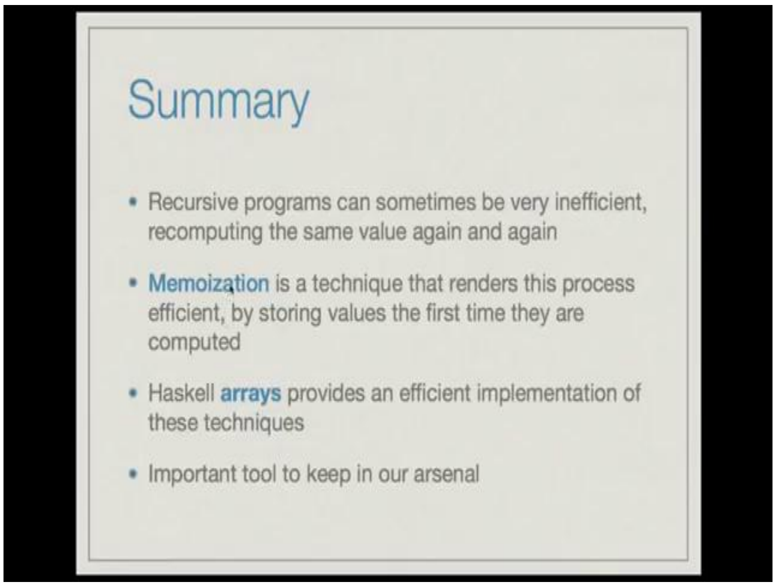
-
  ```
  counts :: [Int] -> [(Int,Int)]
  counts xs = assocs (
                  accumArray (+) 0 (l,u) (zip xs ones)
                  )
     where
        ones = iterate id 1
        l    = minimum xs
        u    = maximum xs
  ```

-
  ```
  arraysort :: [Int] -> [Int]
  arraysort xs = concat [replicate n i | (i,n) <- ys]
     where
        ys   = counts xs
  ```

Finally, here is the code for linear time sorting with accumArray is start to the function count it takes the list of integer and produces the list of pairs of integers this is the associated list of counts, counts of x s is assoc of accumArray with plus 0 l comma u on the association list zip x s ones. Ones, if as you recall is iterate id 1, which produces the infinite list consisting of 1, 1, 1, etcetera, l is the minimum in the original list u is the maximum in the original list with these of indices we produce this array of counts and extract the association list that is embedded in the array into the by using the function assoc, array sort takes y s.

Which is counts of x s and for each entry i comma n y s it does replicate n i and finally, concatenates this list using the function concat. So, finally, you get a sorted version of the original list.

(Refer Slide Time: 49:16)



Summary

- Recursive programs can sometimes be very inefficient, recomputing the same value again and again
- **Memoization** is a technique that renders this process efficient, by storing values the first time they are computed
- Haskell **arrays** provides an efficient implementation of these techniques
- Important tool to keep in our arsenal

In summary, recursive program can sometimes be very insufficient re computing the same value again and again; this is illustrated in the fib function as well as in the lcss function memorization is an important technique that renders this process efficient. Sometimes by storing values the first time they are computed and referring to the store values rather than re computing the in the subsequent times with they are needed. Haskell arrays provide an efficient implementation of these techniques and it is an important tool to keep in our arsenal.