

Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

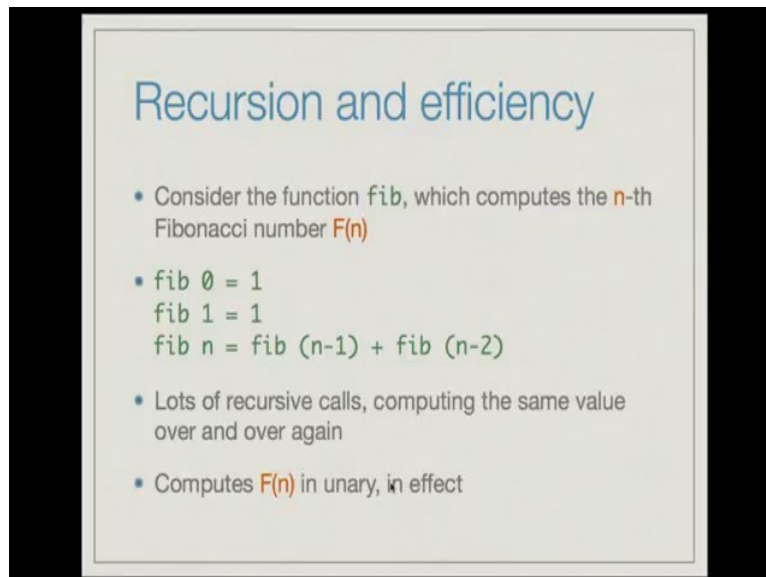
Module # 07

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 $F(n)$
- $\text{fib } 0 = 1$
 $\text{fib } 1 = 1$
 $\text{fib } n = \text{fib } (n-1) + \text{fib } (n-2)$
- 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 will illustrate this with the following example. Consider the function `fib`, which computes the n th Fibonacci number $F(n)$. `fib 0` equals 1, `fib 1` equals 1, `fib n` equals `fib (n-1) + fib (n-2)`. You can see that this program is a straightforward translation of the mathematical definition of the Fibonacci series.

The program is quite simple, but the problem with it is that there are lots of recursive calls computing the same value over and over again. For instance, `fib 3` makes two calls, one to `fib 2` and another to `fib 1`. `fib 2` in turn makes a recursive call to `fib 1` and `fib 0`. So, you see that already `fib 1` is being called twice, if you consider a call like `fib 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 $F(n)$ 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 $G(n)$ be the number of recursive calls to `fib 0` in the computation of `fib` of n for $n > 1$ it is easy to see that $G(2)$ equals 1, because `fib` of 2 = `fib` of 1 + `fib` of 0. So, there is one recursive call to `fib 0`, it is also easy to see that G of 3 equals 1, because `fib` 3 = `fib` 2 + `fib` 1, `fib` of 2 makes a 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 $G(n)$ equals $F(n-2)$, for a proof we see that the statement is true for $n = 2$ and $n = 3$ for $n > 3$, $G(n)$ equals $G(n-1) + G(n-2)$. Because there is one call to `fib` of $n-1$ and one call to `fib` of $n-2$ in `fib` of n and there are $G(n-1)$ calls to `fib 0` in `fib` $n-1$ and $G(n-2)$ calls to `fib 0` in `fib` of $n-2$. But, by induction hypothesis we know that $G(n-1)$ equals $F(n-3)$ and $G(n-2)$ equals $F(n-4)$.

Therefore, $G(n)$ equals $F(n-3) + F(n-4)$, which is $F(n-2)$, which is a large number, which grows exponentially in the size of n . Thus, we see that there are exponentially 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 languages, 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-1} . You initialize the array by saying $fibs[0] = fibs[1] = 1$, then you fill in entries in the array from index 2 till index $n-1$.

So, you initialize $i = 2$, while i is less than n , you just compute $fibs[i]$ to be $fibs[i-1] + fibs[i-2]$. This computation does not involve a recursive call, rather it just picks up 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 n th entry in the array; this program actually takes time proportional to n rather than proportional to 2^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;
```

We 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 `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:...
→ ...
```

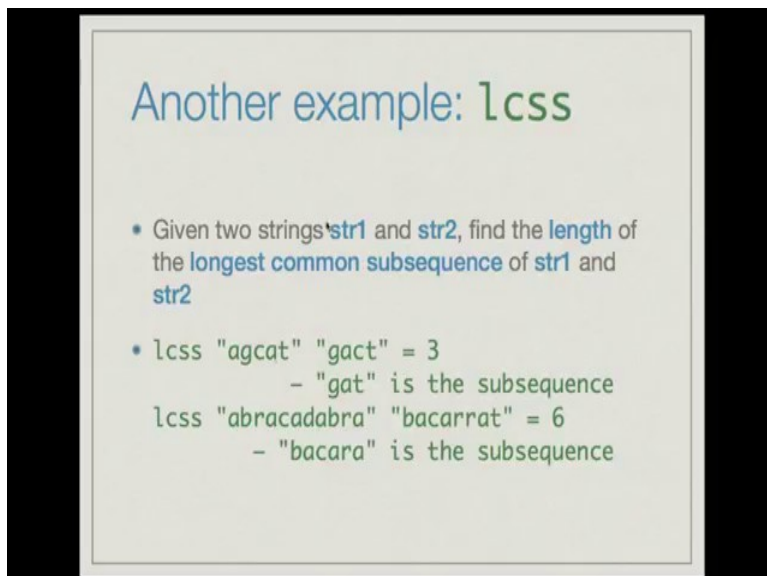
We can also program a linear time Fibonacci function in Haskell by using the power of laziness. Here is the function `fastfib n`, which just builds the Fibonacci series in a list and extracts the `n`th entry, `fibs` is a function with signature `list of integer` and it is given by this

program `fibs = 1 : 1 : zipWith(+) fibs (tail fibs)`. The way the computation unfolds is as follows, this is the expression `1:1: zipWith (+) fibs (tail of fibs)`.

But, `fibs` we know now is `1,1,...` some other entries, `tail of fibs` is `1,...` some other entries. So, now, `zipWith` will add this `1` with this `1` and do a `zipWith` of `+` on the tail of the two lists. 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 `1:1:2: zipWith using (+) [1,2,...]` and the tail of this list which is `[2,...]`

This in turn will give rise to `1:1:2:3: zipWith (+)` on the two lists, `[2,3,...]` and its tail, which is `[3,...]` and so on. You see that you finally, end up with the list `1:1:2:3:5` 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, `str1` and `str2`. We in fact, want to just compute the length of the longest common sub sequence of `str1` and `str2`. For instance `lcss "agcat" "gact" = 3`, because "gat" is a subsequence of "agact" and "gat" is also a sub sequence of "gact", that is the longest common subsequence, `lcss` of "abracadabra" and "bacarrat" equals 6, because "bacara" 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  $\geq 2^n$ , 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,  $\text{lcss} \text{ "" } \_ = 0$ . lcss of the empty string with anything is 0, because the empty string does not have any subsequence. Or you could say the empty string has only itself as a subsequence and the empty string is always a subsequence of any other sequence.  $\text{lcss} \_ \text{ ""} = 0$ . lcss of anything else and the empty string is also 0, lcss of two non empty strings, which are given by (c:cs) and (d:ds), is computed as follows.

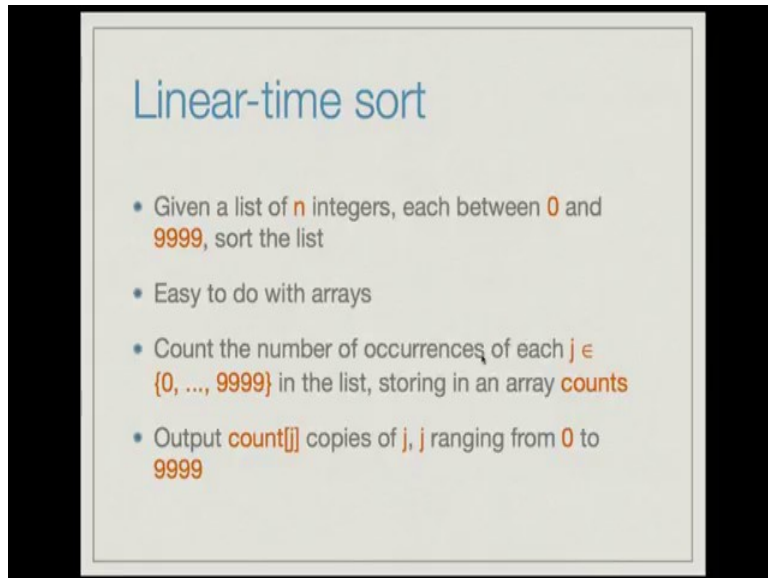
In the case that c is equal to d, then you know that any subsequence any common sub sequence of cs and ds can be extended by adding c to the front and that will give you a subsequence of length one longer. So, if you have something to be the longest common subsequence of cs and ds, which is computed by  $\text{lcss cs ds}$ , 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  $\text{lcss cs ds}$ .

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 subsequence of (c:cs) and (d:ds) is either the longest common subsequence of (c:cs) with ds or the longest common subsequence of cs with (d:ds) which is exactly, what we are doing here. In the case that c is not equal to d, the longest common subsequence, the length of the longest common subsequence of these two lists is the same as max of ( $\text{lcss (c : cs) ds}$ ) and ( $\text{lcss cs (d : ds)}$ ).

One can prove that lcss of cs and ds, which are two strings takes time at least  $2n$ , where cs and ds 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 out 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)



Here is another example, which is that of linear time sort, we already know that programs like merge sort take  $O(n \log n)$  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 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  $\text{count}[j]$  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 ranging 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-1, you look at each array entry and then update the appropriate counts entry.

So, if arr[i] equals 500 let say, counts[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  $O(n+10000)$ , so if n is much larger than 10000, we have actually managed 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 achieve 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  $i$ th 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.Array`.

(Refer Slide Time: 14:23)

## Arrays in Haskell

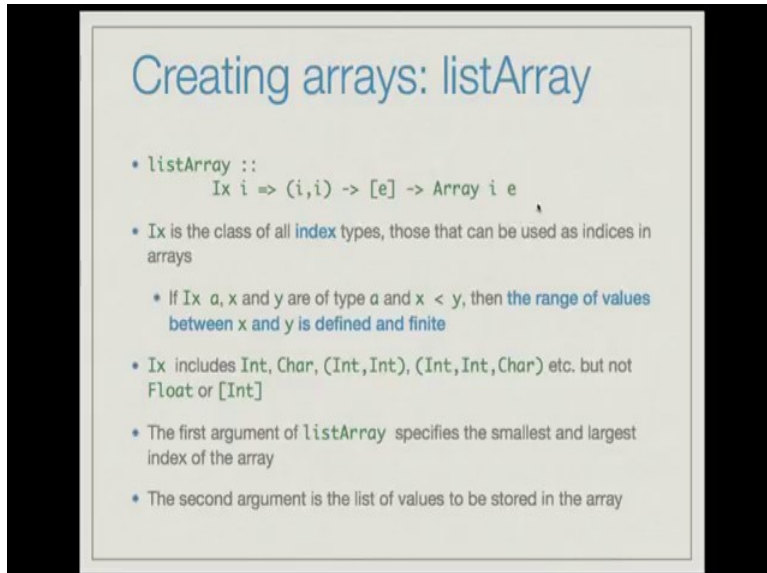
- `import Data.Array`  
`myArray :: Array Int Char`
- The **indices** of the array come from `Int`  
The **values** stored in the array come from `Char`
- `myArray = listArray (0,2) ['a','b','c']`

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

Here is how we use this `import Data.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, if you say that `myArray` equals `listArray (0,2) ['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 `listArray` in more detail now. `listArray` has type `Ix i` implies the pair `(i,i) -> [e] -> Array i e`. `Ix` is the type class of all index types, which are those types that can be used as indices in array. If `Ix a` holds and `x` and `y` are of type `a` and `x < y`, then the range of values between `x` and `y` is defined and finite. This is the property of the type class `Ix`. For instance `Ix` includes the types `Int`, `Char`, `(Int, Int)`, the triple `(Int, Int, Char)` etc, but not `Float` or `[Int]`.

Because, if you take two floating point values `x` and `y` and let, say `x < 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 many lists that lie in between these two. The list consisting of `[1,2]`, the list `[1,1,2]`, the list `[1,1,1,2]` etc. All lie in between the singleton list 1 and the singleton list 2, therefore, the list `Int` cannot be used as an index type.

The first argument of the `listArray` function specifies the smallest and largest index of the array `(i,i)`, the second argument is the list of values to be stored in the array. And finally, the output is an array whose index type is `i` and whose value type is known.

(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` applied to the pair `(1,1)` and the list `[100...199]` will give you the following array, `array (1,1)`, which is the range of the indices, the indices range from 1 to 1, which means that there is only one index. And the list itself, the array itself consists of one entry with index 1 and value 100. The values in the array are filled in the order they are presented in the list, so therefore 100 is associated with the index 1 from and not say 129.

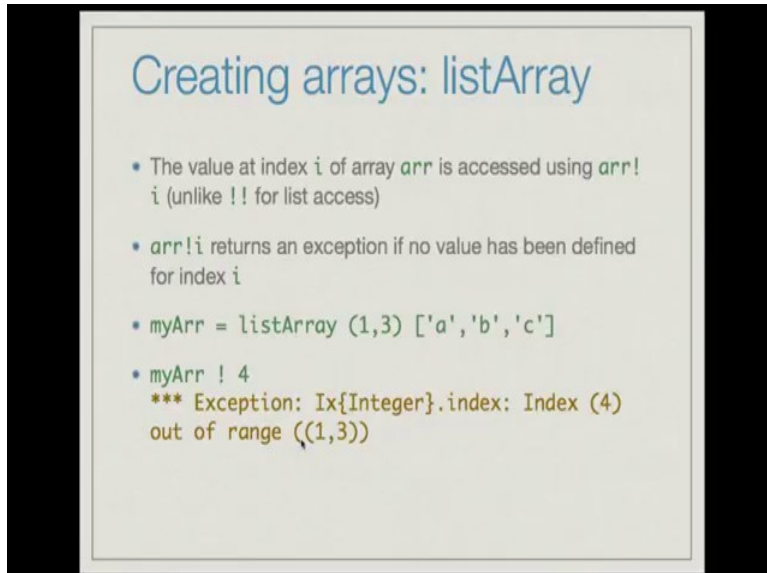
Here is an another example. `listArray` the pair `('m','p')` where `m` and `p` are the characters and, where you want to store values from `[0,2,..]`, which is the infinite list of all the even positive integers will give you the following array. Array with the index bounds `m` and `p` and the entries of the array, there are 4 entries in the array, in the index `m` stores the value 0 ,index `n` stores the value 2, index `o` stores the value 4, index `p` stores the value 6. Here is another example, `listArray ('b','a')` and values from `[1..]`, which is the infinite list will give you the empty array.

This is because, `'a'`, the character `'a'` is actually less than the character `'b'`. So, therefore, there cannot be any index. The prerequisite for an 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(0,4). [100..]`, this will be array with bounds for the index 0 and 4 and entries being `(0 , 100), (1,101), (2,102),(3,103)` and `(4, 104)`.

Here is another example `listArray (1,3)`, which tells that there are three indices 1 2 and 3 and the list itself has only two elements `['a','b']` this will actually produce an exception. Because

Haskell tries to fill a value corresponding to index 3, but it cannot find 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 an array is accessed using the single exclamation mark like so, `arr ! i` (unlike the double exclamation mark for list access). So, `arr ! i` returns the `i`th value in the array, but it returns an exception if no value has been defined for index `i` which is why in the earlier slide, we saw that we got an exception for creating an array with three index values, but one with only two values.

Suppose, we created an array, `myArray` using `listArray (1,3), ['a','b','c']`, now if you try to access the fourth element in the array, `myArr ! 4` will again get an exception saying that 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  $i \geq 2$
- `listArray` takes time proportional to the range of indices,

An important point to note is that 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 the array and define the  $i$ th element of the array in terms of the  $i$  minus first element and  $i$  minus second element. This is exactly the reminiscent of the array version of the computing the Fibonacci series in c, another point to note is that `listArray` takes time proportional to range of the indices. So, if there are  $k$  indices the call to `listArray` takes time  $O(k)$ .

(Refer Slide Time: 22:15)

## Creating arrays: array

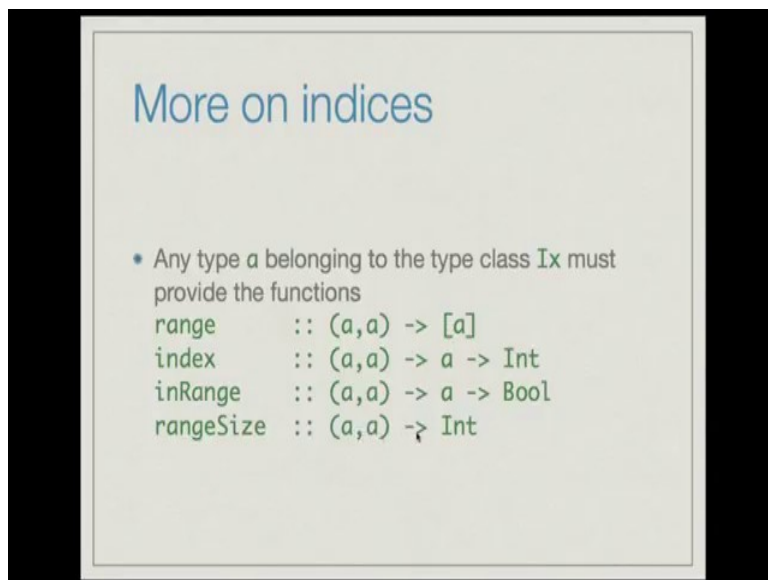
- `array ::`  
`Ix i => (i, i) -> [(i, e)] -> Array i e`  
Creates an array from an associative list
- The associative list need not be in ascending order of indices  
`myArray = array (0,2)`  
`[(1,"one"),(0,"zero"),(2,"two")]`
- The associative list may also omit elements  
`array (0,2) [(0,"abc"),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

There is another way to create arrays, which is to use the function `array` whose type is as follows `Ix i => (i,i) -> [(i,e)] -> Array i e`. So, instead of producing a list of only values,

which was a list of elements of type `e`, we are now providing an associative list and creating an array out of this.

The associative list need not be in ascending order of the indices, for instance you could create an array as follows `myArray = array (0,2) [(1,"one"), (0,"zero"),(2,"two")]`. But, notice that the elements of the associative list are not presented in ascending order of the indices. The associative list may also omit elements, so you have `array (0,2)[(0,"abc"), (2,"XYZ")]` there is no entry for 1, this call also takes time proportional to the range of the indices.

(Refer Slide Time: 23:37)



Let us look at indices in a little more detail. any type `a` belonging to the type class `Ix` must provide the following functions. A range, which is a function from pairs of the form `(a,a) -> [a]`, index, which is a function with signature `(a,a) -> a -> Int`. function `inRange`, which has signature `(a,a) -> a -> Bool` and the `rangeSize`, which has signature `pair (a,a) -> 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
 index gives 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, `range :: (a,a) -> [a]`, gives the list of indices in the sub range defined by the bounding pair, the first a is a lower bound and the second a is upper bound and this list gives all the elements that lie between the first element here and the second element here. For instance `range (1,2)` is the list `[1,2]`. `range('m','p')` is the string "mnop". `Range ('z', 'a')` is the empty string, because a is smaller than z.

Index gives the position of the subscript in the subrange, for instance `index` of -50 in the subrange defined by `(-50,60)` is 0, `index` of 35 in the sub range defined by `(-50,60)` is 85. Because, you go from 50 to 0 and then from 1 to 35, `index` of o the character 'o' in the range defined by the character 'm' and character 'p' is 2, because you go m is 0 n is 1 and the o is 2. `Index` of a, the character a in the range defined by `('m','p')` will give an exception, because 'a' is out of the range `('m','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`

`inRange` is a function that checks whether a given element lies in the range or not. for instance `inRange` of `(-50,60)` applied on `(-50)` will be true. You can also check that `o` is inside the range defined by `m` and `p`, but `a` is not inside the range defined by `m` and `p`, so `inRange` returns false. `rangeSize` just gives you the size of the range defined by the lower bound and the upper bound, for instance `rangeSize` applied to `(-50,60)` will give you 111. Range size applied to the character range `('m','p')` will give you 4. `rangeSize` defined by `(50,0)` 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

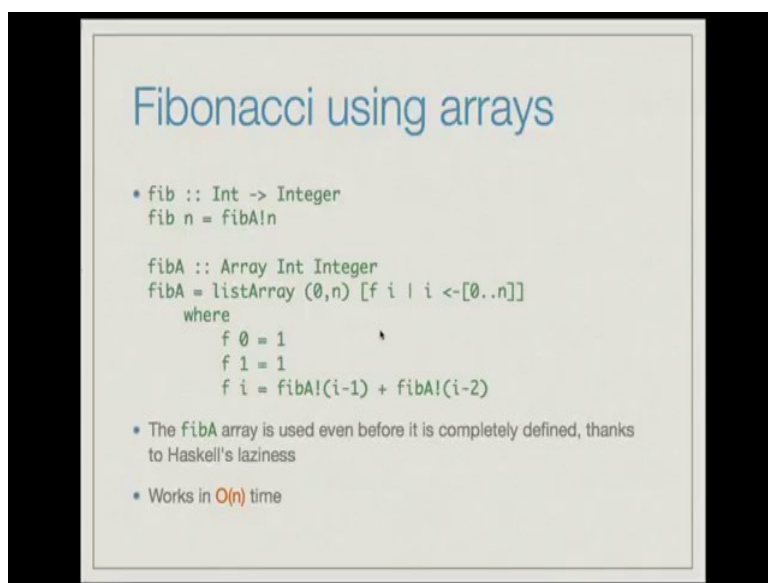
Here are some more function on arrays. `(!)` the exclamation mark which we mentioned earlier is to access an array entry, it has signature `Ix i implies Array i e -> i -> e`, so it gives the value



at a given index in an array. Bounds is a function that takes Array i e as an argument and returns the lower and upper bounds of the indices. The bounds with which the array was originally constructed. Indices is a function that lists the indices of an array in ascending order rather than providing just a pair, it produces the list of all indices.

Elms is the function that provides produces the list of all elements of an array in index order. Assocs is a function that lists all associations of array in the ascending order of index, its signature is  $I \times i \Rightarrow \text{Array } i \ e \rightarrow [(i,e)]$ .

(Refer Slide Time: 27:42)

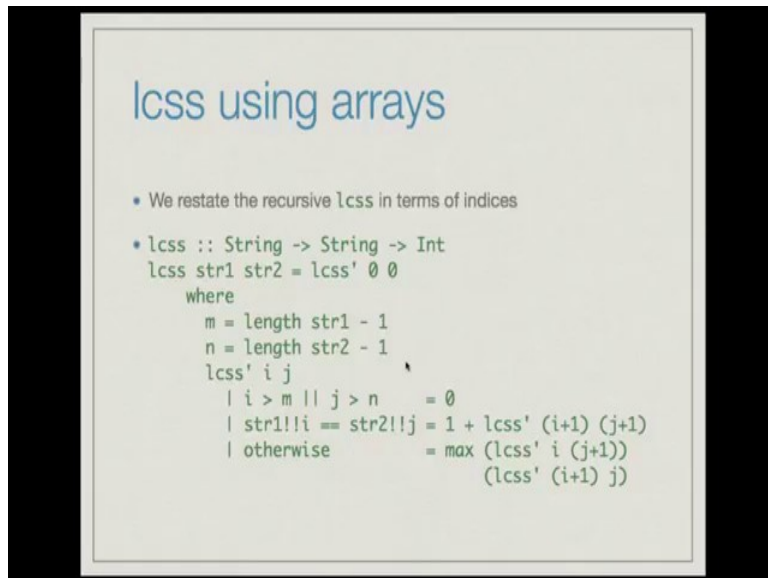


Let us now, get back to our original example that of computing Fibonacci numbers, but now using arrays. Fib is a function with signature  $\text{Int} \rightarrow \text{Integer}$ . fib of n equals fibA!n, which is accessing the nth element of the array fibA, fibA is Array Int Integer the indices are from Int and the values are from Integer. And now we use the fact that Haskell arrays are lazy. fibA is defined to be listArray (0,n), where the ith entry is given by the function f i.

You create an array out of the list consisting of all values of f i. f i for i ranging from 0 to n. f is defined as follows f of 0 is 1. f of 1 is 1. f of i is not f (i-1) + f (i-2), as we would have done in a recursive program. But, fibA!(i-1) + fibA!(i-2), we are just referring 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 fibA has not be defined yet, Haskell will lazily make a call to f (i-1).

But, the first time it will make a call to `f`, it will fill the entry in the array, but the second time the called to `f (i-1)` is made it will just pick out the entry from the array. The `fibA` array is used even before it is completely defined, thanks to Haskell's laziness and this program works in  $O(n)$  time, because all it needs is to just fill  $n$  indices, by referring to previous entries in the array.

(Refer Slide Time: 29:43)



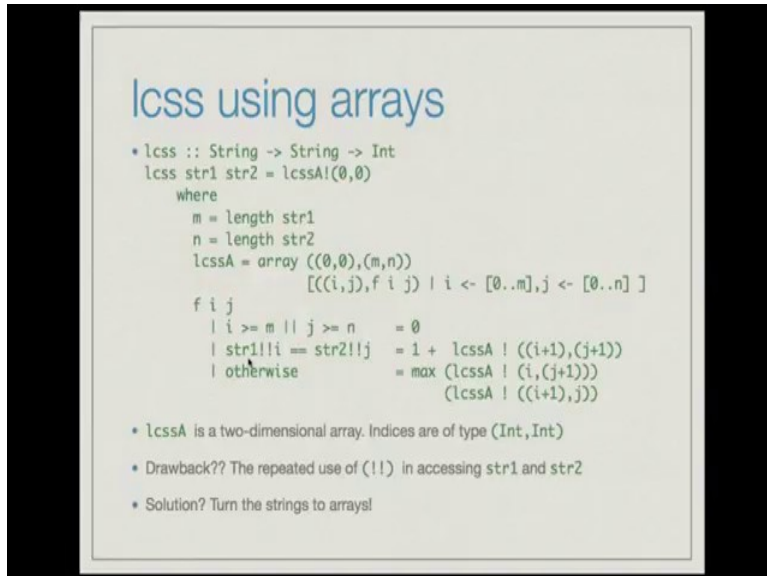
Here is how we can compute lcss using arrays, we restate, we first restate the recursive lcss in the terms of indices. lcss is a function with signature `String -> String -> Int`. But, we will work with the version `lcss'`, which works as `Int -> Int -> Int`. `lcss str1` and `str2` equals `lcss' 0 0`, where `lcss'` computes the length of the longest common sub sequence of drop `i` `str1` drop `j` `str2`.

And since, drop 0 of `str1` is equal to `str1` and drop 0 of `str2` is equal to `str2`, computing lcss prime of 0, 0 achieves, what you want to achieve, which is to compute lcss of `str1` and `str2`. Let `m` be the length of `str1 - 1` and `n` be the string length of `str2 - 1`. these are the last indices, if you will of `str1` and `str2`. Now, lcss prime of `ij` is defined as defined as follows. if `i` is greater than `m` or `j` is greater than `n` the value is 0.

Otherwise if the character at the `i`th location of `str1` is the same as character at the `j`th location of `str2`, the result is `1 + lcss'` of `(i+1) (j+1)`. If, the `i`th character of `str1` is not equal to the `j`th character of `str2`, then as earlier we compute the result using max of (`lcss' i (j+1)`) and (`lcss'`

(i+1) j). Now, we have restated the original lcss function in terms of recursion on indices, this we will try to transcribe directly into an array based program.

(Refer Slide Tim-e: 32:07)



Here is lcss using arrays, lcss of str1 str2 is the 0 0'th entry of the array lcssA. the array lcssA is a two dimensional array whose indices range from (0, 0) to (n,n). The entry of the array at (i,j) is supposed to be the longest common sub sequence of str1 starting from index i and str2 starting from index j or in other words drop i of str1 and drop j of str2. In creating lcssA we use the array function rather than the listArray function, because it is easier to provide the values of the array in terms of an associative list.

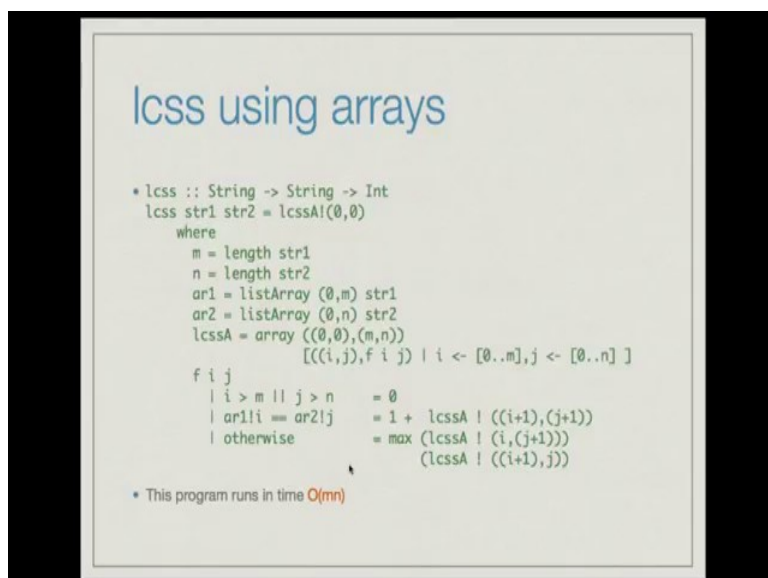
So, we define lcssA as array with range (0, 0) to (n,n) and entry is of the form the part (i,j) comma the value f applied to i and j. So, this is the index and this is a value and the index and value is given as a pair, where i ranges from 0 to n and j ranges from 0 to n. In a two dimensional array of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

Now, f is defined as follows f on i and j is very similar to the index based recursive function that was described earlier, if i is greater than or equal to m or if j is greater than or equal to n the value is 0. Otherwise; if check if the ith element of string1 is the same as the jth element of str2.

In that case the result is  $1 + \text{lcSSA}!((i+1), (j+1))$ . Notice, that here instead of making a recursive call to  $f$  we just refer to the corresponding element in the array at the appropriate index. Otherwise, if  $\text{str1}!!i$  is not equal to the  $j$ th element of  $\text{str2}$ , we define  $f$  of  $ij$  to be  $\max$  of  $(\text{lcSSA}! (i, j + 1))$  and  $(\text{lcSSA}! (i+1, j))$  this is the function.

One minor drawback here is that we repeatedly use the exclamation, exclamation in accessing  $\text{str1}$  and  $\text{str2}$  here. Recall that we said that accessing an element of an array can be done in constant time whereas accessing the  $i$ th element of a list takes time order of order  $i$ , the solution would be to turn the strings themselves into arrays, which is done as follows.

(Refer Slide Time: 35:32)

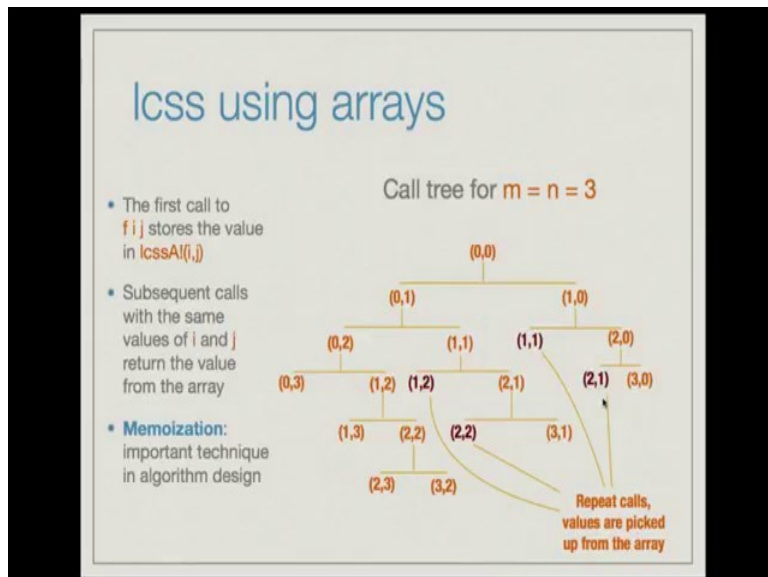


Here is the version, which works on arrays rather than strings.  $\text{lcSS}$  of  $\text{str1}$  and  $\text{str2}$  is  $\text{lcSSA}! (0,0)$ . Where instead of using  $\text{str1}$  and  $\text{str2}$  in the description, of  $f$ , we use  $\text{ar1}$  and  $\text{ar2}$ , which are 2 arrays.  $\text{ar1}$  is got by  $\text{listArray } (0, m) \text{ str1}$ , where  $m$  is the length of  $\text{str}$ .  $\text{ar2}$  is  $\text{listArray } (0, n) \text{ str2}$ , where  $n$  is the length of  $\text{str2}$ .  $\text{LcSSA}$  is itself defined as usual array with indices ranging from 0 to 0 to  $m$  comma  $n$  and entries of the form  $(i,j)$ ,  $f$  applied to  $i$  and  $j$ .

Where  $i$  ranges from 0 to  $m$  and  $j$  ranges from 0 to  $n$ .  $f$  is now defined in terms of  $\text{ar1}$  and  $\text{ar2}$ .  $f$  of  $i j$  equals 0 if  $i > m$  or  $j > n$ . Otherwise; if  $\text{ar1}$  at position  $i$  is equal to  $\text{ar2}$  at position  $j$ , then you define it to be  $1 + \text{lcSSA}! ((i+1), (j+1))$ . Otherwise; you define it as usual. the difference here is that we access the array rather than the string repeatedly, this program one can check runs in time  $O(mn)$ .

Because, all we need do is fill in elements in the array and the number of indices of this array is  $(m+1)*(n+1)$ . So, the program runs in time  $O(m)$  which is a vast improvement over  $2^n$  that we had earlier.

(Refer Slide Time: 37:24)



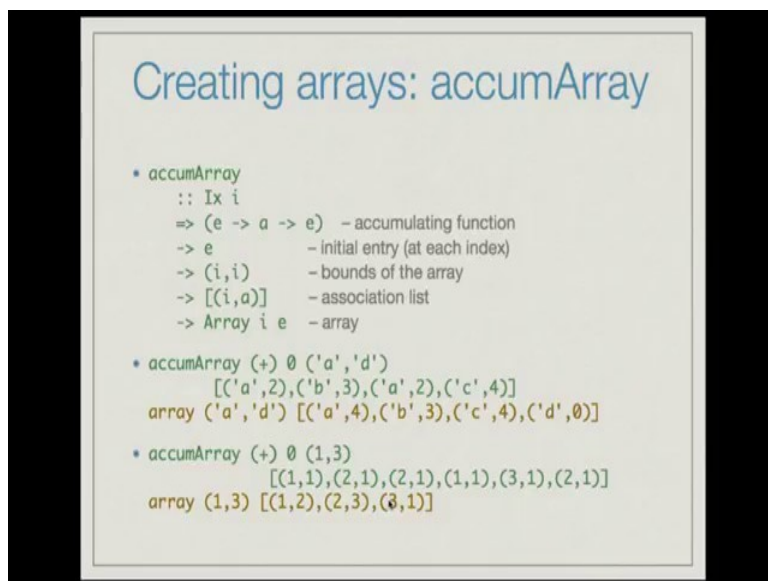
Let us look at the computation in a little more detail, let us look at the call tree for the case when  $m = n = 3$ , the call tree for lcss. So, the first call to  $f(i, j)$  is we are considering  $f$  of 3 3 the first call to  $f$  of  $i, j$  stores the value in the array and subsequent calls with the same values  $i$  and  $j$  return the value from the array rather than making the recursive call and this technique is called memoization, it is an important technique in algorithm design.

Where you translate a recursive algorithm to a more efficient version by storing the values and referring to them later. So, initially there is a call made to  $f$  of  $(0, 0)$ . This points calls to  $f$  of  $(0, 1)$  and  $f$  of  $(1, 0)$  possibly it might also span a call to  $f$  of  $m+1$ , but that occurs here anywhere.  $f$  of  $(0, 1)$  might span calls to  $f$  of  $(0, 2)$  and  $(1, 1)$ .  $(0, 2)$  spans calls to  $(0, 3)$   $(1, 2)$  this in turn leads to calls.  $(0, 3)$  terminates, because 3, if you recall is greater than or equal to  $n$ , which is 3.

Therefore, this call terminates by giving the value 0 and that will be stored in the array.  $(1, 2)$  gives rise to calls to  $(1, 3)$  and  $(2, 2)$ .  $(2, 2)$  makes call to  $(2, 3)$  and  $(3, 2)$ . These two calls terminate, because here 3 is greater than  $n$ . Now, once this returns we will have to process calls to  $(1, 1)$ ,  $(1, 1)$  gives rise to calls to  $(1, 2)$ ,  $(2, 1)$  but  $(1, 2)$  here is a repeat call. So, the values are picked up from the array rather than leading to a recursive call.

So, there won't be any, this whole tree under (1,2) will not be repeated. (1,1) also gives rise a call to (2,1) , which in turn gives rise to calls to (2,2) and (3,1) but (2,2) is a repeat cal. It occurs earlier here already. So, its values are picked up from the array and this tree is not repeated under, (3,1) terminates, because 3 is greater than or equal to n and so on, you see that there are two more repeat calls here. So, in this manner we see that we never have to spend time more than  $O(n)$  and this example also illustrates, how exactly lazy arrays work.

(Refer Slide Time: 40:13)



Here is another way to create an array, which is to use the function called accumArray, accumArray accumulates values into array positions and it works as follows. Its signature is  $Ix\ i \Rightarrow (e \rightarrow a \rightarrow e)$ , which is an accumulating function. this is similar to the kinds of functions you would provide as arguments to foldr or foldl. In particular this is the kind of function that you would provide as an argument to foldl, e which is an initial entry that will be placed in each index of the array (i,i) this provides the bounds of the array.

And an association list which is a list of pairs of (i,a), with all this it produces an array Array i e. Lets look at how it 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 associative list given here, [( 'a',2), ( 'b',3), ( 'a',2), ( 'c',4)] produces the following array. An array with indices ranging from 'a' to 'd' and entries [( 'a',4), ( 'b',3), ( 'c',4), ( 'd',0)] .

How do you explain the entries? Well 4 is 2+2, 3 is just 3, the 4 here is just 4 and the 0 here, is the initial value that was 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 a 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 was provided is plus, if it is star it would have 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. Look at another example `accumArray` with function (+) and initial value 0 and the bounds (1,3) and the list provided by [(1,1), (2,1),(2,1),(1,1),(3,1),(2,1)] produces the following array, array bounds (1,3) and entries being [(1,2), (2,3), (3,1)].

Here we are just counting the number of repetitions of each element 1 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 plus to add 1 every time, we encounter 1. Similarly, whenever we encounter 2 we add 1 to the accumulator. Since 2 occurs thrice here, here and here and the final value associated to index 2 in the array is 3, 3 occurs once, so that finally, you will have one as the value at the 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,u) list`. `f` is the function `e` is the initial entry `(l,u)` is the lower bound on the indices and upper bound on the indices and the list is the associated list. It creates an array with indices `l..u` in time proportional to `u - l` which is important provided `f` can be computed

in constant time. So, for a particular  $i$  between  $l$  and  $u$ , if  $(i,a_1), (i,a_2), \dots (i,a_n)$  are all the elements with index  $i$  appearing in this list.

The value for  $i$  in the array is  $f(\dots(f(f e a_1) a_2) \dots) a_n$ .  $f$  applied to  $e$  and  $a_1$ .  $f$  applied to the first result and  $a_2$ ,  $f$  applied to the second result and  $a_3$ , etcetera and finally,  $f$  applied to that result and  $a_n$ . So, it is a foldl 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] [1,1,1,1,1,1,1,1,1,1,1]`
- = `[(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 = [1,1,1,1,1,1,1,1,1,1,1]`)
- `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 in linear time as follows, suppose we are given the list 2, 3, 4, 1, 2, 5, 7, 8, 1, 3, 1. We first create another list an association list as follows by zipping it with 1 repeated infinitely often that will produce a list of pairs where the first element of each pair is from the original list and the second element is always 1.

How do you produce this list `[1,1,1,...]`, recall that `iterate` is a function, which behaves as follows: `iterate f x` equals the list `[x, f x, f (f x), f (f (f x)) ...]`. 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 1s. 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

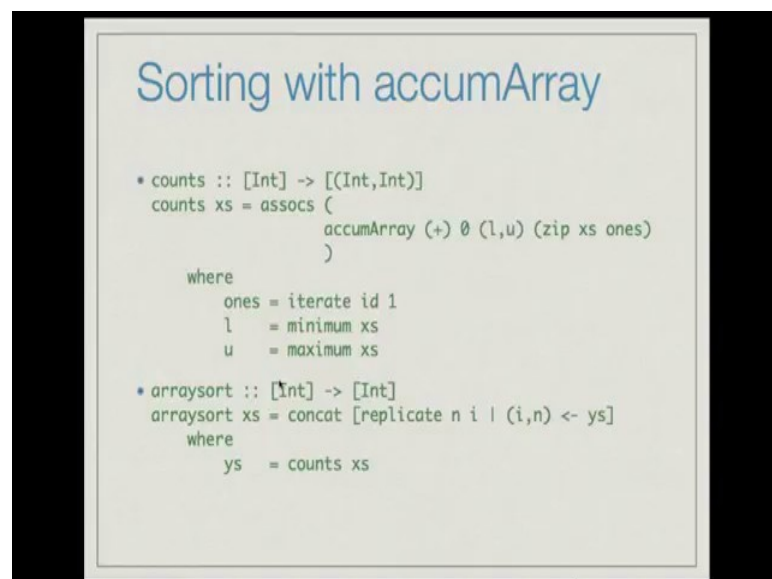


array you go to the list [(1,3),(2,2),(3,2),...]. You can obtain this list from the array by using the function `assocs`. Once you have this list, you replicate 1 thrice and that is done using the function `replicate`, `replicate n i` gives you the list consisting of the value `i` repeated `n` times.

So, `replicate 3 1` gives of the list `[1,1,1]`. When you replicate 2 twice, `replicate 2 2` .replicate of 2 3 that is, because you want to replicate the value 3 twice. `replicate 1 4 ++ replicate 1 5` etcetera. That will produce this list `[1, 1,1] ++ [2, 2] ++ [3,3] ++` 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 `assocs` also works in time proportional to the length of the list produced. Therefore, this algorithm works in linear time or order `n` plus `max` minus `min` of the elements in the array, if you prefer of the elements in the list.

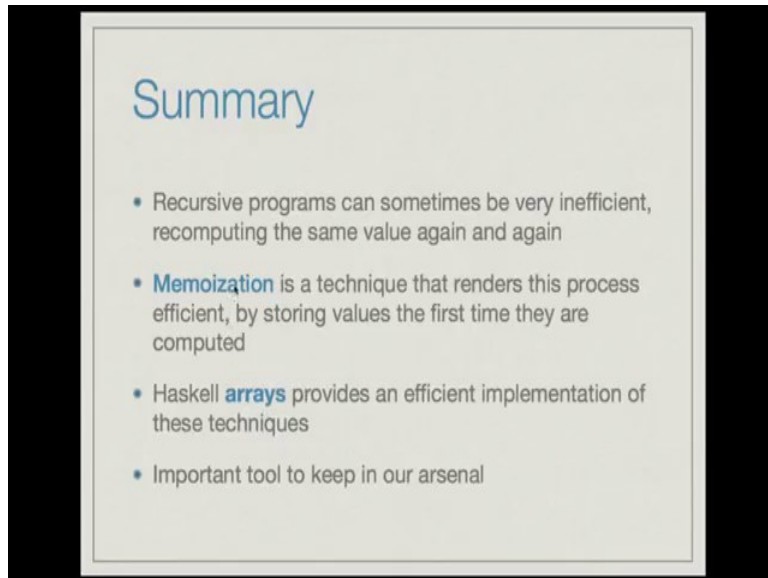
(Refer Slide Time: 48:05)



Finally, here is the code for linear time sorting with `accumArray`, you start with 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 xs` is `assocs of accumArray with (+) 0 (l,u)` on the association list `zip xs 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 as indices we produce this array of counts and extract the association list that is embedded in the array by using the function `assocs`,

Arraysort takes `ys` which is counts of `xs` and for each entry  $(i,n) \leftarrow ys$ , 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)



In summary, recursive program can sometimes be very inefficient, re computing the same value again and again; this is illustrated in the `fib` function as well as in the `lcs` function, memoization 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 in the subsequent times when they are needed. Haskell arrays provide an efficient implementation of these techniques and it is an important tool to keep in our arsenal.