INDIAN INSTITUTE OF TECHNOLOGY MADRAS
सिद्धिर्भवति कर्मजा

IIT Madras

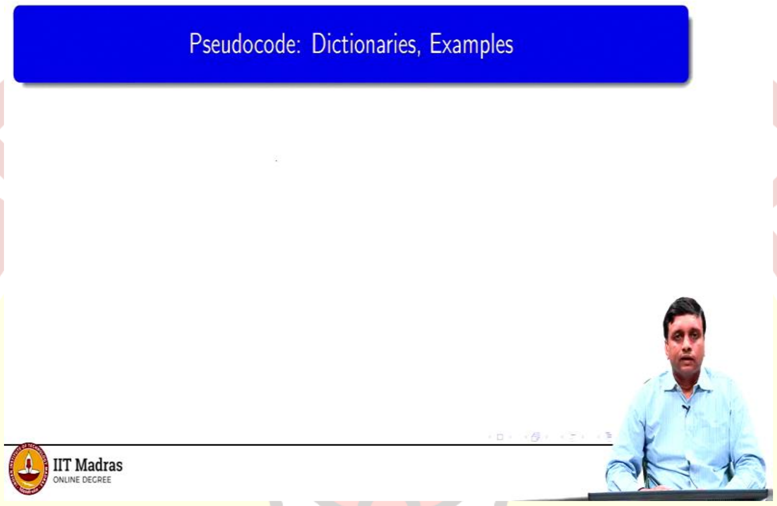
ONLINE DEGREE

`Computational Thinking

Professor Madhavan Mukund
Department of Computer Science
Chennai Mathematical Institute
Professor G. Venkatesh
Indian Institute of technology, Madras

Pseudocodes for Real-Time Examples Using Dictionaries

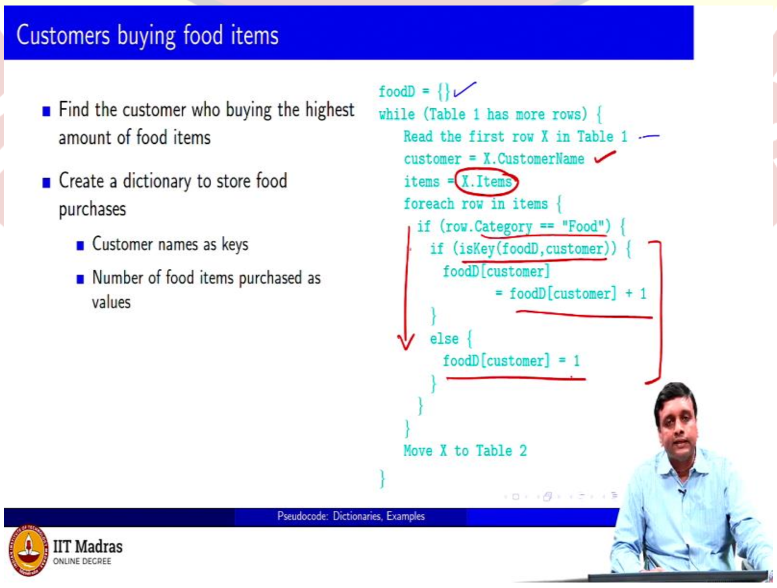
(Refer Slide Time: 00:14)



Pseudocode: Dictionaries, Examples

Now, that we have augmented our pseudocode with notation for dictionaries. Let us work through some of the examples, that we saw in the class and see how it looks in terms of pseudocode.

(Refer Slide Time: 00:24)



Customers buying food items

- Find the customer who buying the highest amount of food items
- Create a dictionary to store food purchases
  - Customer names as keys
  - Number of food items purchased as values

```
foodD = {}
while (Table 1 has more rows) {
    Read the first row X in Table 1
    customer = X.CustomerName
    items = X.Items
    foreach row in items {
      if (row.Category == "Food") {
        if (isKey(foodD,customer)) {
          foodD[customer]
                = foodD[customer] + 1
        }
        else {
          foodD[customer] = 1
        }
      }
    }
    Move X to Table 2
}
```

So the first example we will look at, comes from the shopping bills data. So, in the shopping bills data, we had item information about each shopping bill of across customers, and different shops. So, suppose we wanted to find a customer, who buys the highest number of food items. So, we are just counting the highest number of food items. So, one way to do this is to create a dictionary, and in this dictionary we will store the quantity of food items purchased by each customer.

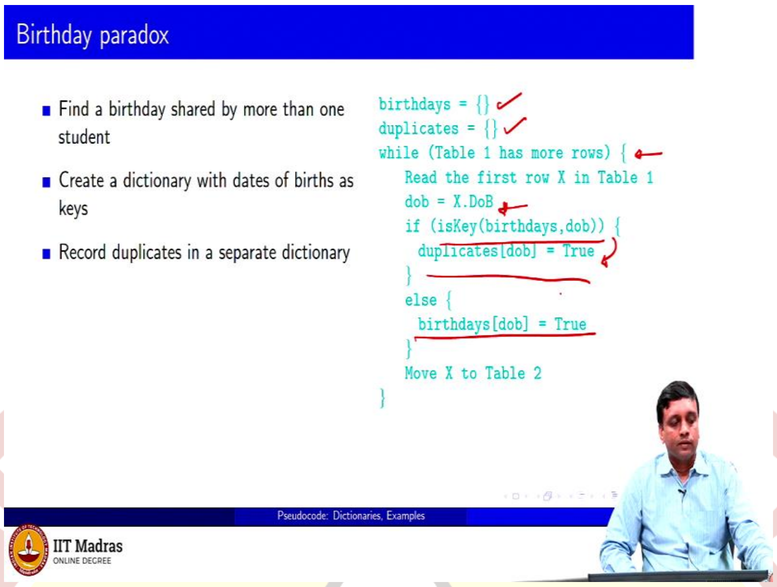
So, what we want here is to accumulate, we want to accumulate against each customer's name, how many food items that person has bought. So, the keys will be the customer names, and the values will be the number of food items across all the bills of that customer. So here is the pseudocode for this. So we first start with an empty dictionary, food dictionary, and we now go through the shopping bills data row by row. So, while there are more rows, we read each row. Now remember that within a row, the shopping items are themselves a list.

So, we pick up the name of the customer, and then we pick up the items that are there against that customer in that bill. And we assume, let us say that, this is a nested table in some sense. So we go through each row in that table. So what we want to now check is, whether an item in that table of items that this person has bought in this bill is a food item or not. So, if the category is food, then I need to update the dictionary for this customer.

So, as we had seen, when we introduce a notation for dictionary, we have to decide at this point whether or not there is already entry for this customer. If there is already an entry for the customer, then we have to increment the number of food items bought by the customer. If there is no entry, then we have to create an entry and say this is the first food item. So that is what this if condition is saying. It is saying, if there is already a key in this food dictionary for this particular customer, then increment the entry for that customer by 1. So, we add 1 more to it. Otherwise, create a new entry for this customer and proceed.

So, this is a typical example as we said, where a dictionary is accumulating values. The values to be accumulated are attached to some keys, in this case the names of the customers. But, we do not know in advance which keys are present and which keys are not present. There are many customers in that shopping bill data set who may not have bought food items at all. And there is no need to create an entry for them if they have not bought food item at all. Because, we are looking for, we know that some people have bought food items. What we are finally interested in is finding out the maximum thing. So we have not shown the second part of this, which would be to go through all these entries and then look for the maximum.

(Refer Slide Time: 03:08)



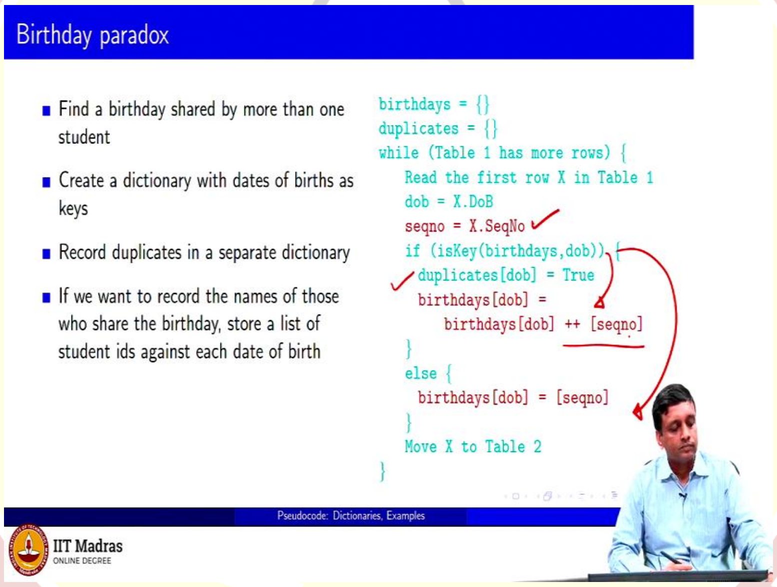
Birthday paradox

- Find a birthday shared by more than one student
- Create a dictionary with dates of births as keys
- Record duplicates in a separate dictionary

```
birthdays = {}
duplicates = {}
while (Table 1 has more rows) {
    Read the first row X in Table 1
    dob = X.DoB
    if (isKey(birthdays,dob)) {
      duplicates[dob] = True
    }
    else {
      birthdays[dob] = True
    }
    Move X to Table 2
}
```

Pseudocode: Dictionaries, Examples

IIT Madras ONLINE DEGREE



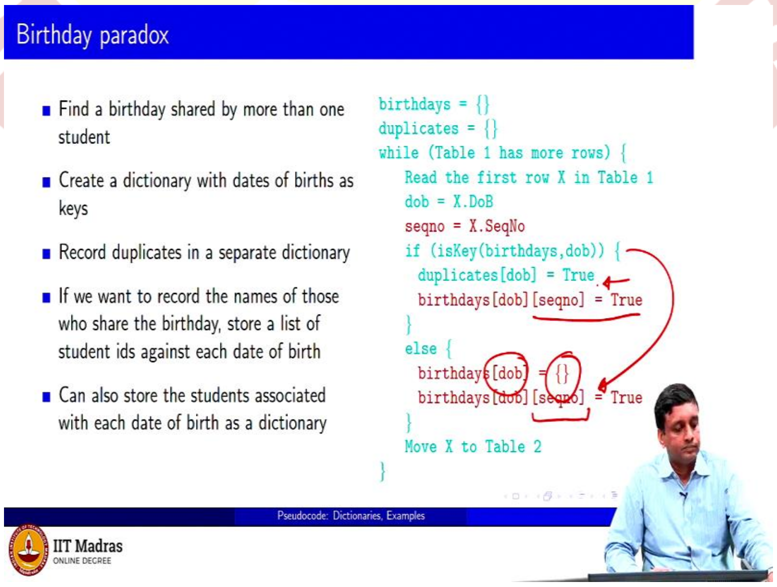
Birthday paradox

- Find a birthday shared by more than one student
- Create a dictionary with dates of births as keys
- Record duplicates in a separate dictionary
- If we want to record the names of those who share the birthday, store a list of student ids against each date of birth

```
birthdays = {}
duplicates = {}
while (Table 1 has more rows) {
    Read the first row X in Table 1
    dob = X.DoB
    seqno = X.SeqNo
    if (isKey(birthdays,dob)) {
      duplicates[dob] = True
      birthdays[dob] =
         birthdays[dob] ++ [seqno]
    }
    else {
      birthdays[dob] = [seqno]
    }
    Move X to Table 2
}
```

Pseudocode: Dictionaries, Examples

IIT Madras ONLINE DEGREE



Birthday paradox

- Find a birthday shared by more than one student
- Create a dictionary with dates of births as keys
- Record duplicates in a separate dictionary
- If we want to record the names of those who share the birthday, store a list of student ids against each date of birth
- Can also store the students associated with each date of birth as a dictionary

```
birthdays = {}
duplicates = {}
while (Table 1 has more rows) {
    Read the first row X in Table 1
    dob = X.DoB
    seqno = X.SeqNo
    if (isKey(birthdays,dob)) {
      duplicates[dob] = True
      birthdays[dob][seqno] = True
    }
    else {
      birthdays[dob] = {}
      birthdays[dob][seqno] = True
    }
    Move X to Table 2
}
```

Pseudocode: Dictionaries, Examples

IIT Madras ONLINE DEGREE

So, as our second example, let us look at this birthday paradox. So, this birthday paradox says that if there are more than about 23 people in a group, then there is a 50 percent chance at least 2 of them have the same birthday; that is the same date of birth and same month of birth. So, we tested this out on our classroom data set, the scores data set. So, we wanted to check whether there is a birthday which is shared by 1, more than 1 student, and of course a naive way to do this is to use nested loops.

You take 1 student and then you compare this student's birthday with every other student, and you do this across all students. So, we have 2 nested loops; for every student in the class, check for every other student in the class whether they have the same birthday or not. We also said that, we could group these, we could bin these, by month of birth and reduce this nested loop and so on.

So now let us see, how a dictionary can help us solve this problem very efficiently. So what we will do is, we will create a dictionary in which we will record all the dates of birth that appear in our data set. So for every date of birth which appears, we will create an entry in the dictionary and whenever we see a new card, or a new row in the table, and the date of birth is already present in our data set, we can say that this data set is, this date of birth is a duplicate.

So, we start with 2 dictionaries, the dictionary birthdays will store all the dates for which birthdays exist in our data set, and duplicates will store all those days of birth which occur more than once. So initially, they are both empty, because we have seen nothing. So now, we go through our scores data set, and we pick up the date of birth on the card. So X dot DoB is the date of birth on the current card, and we asked whether it is present or not. Ideally if they are all distinct, it should not be present. So I check whether the date of birth is already being recorded, if it is not being recorded, I create an entry for it.

I do not want to record anything more than the fact, that it exists. So I just store the value, true. So, this is just a simple way of indicating that this. So this is like, accumulating a list of birthdays, but we are not accumulating a list, as we will see, this is a common paradigm, instead of using a list, we are keeping a set of keys recording each item in that list. So this, now is a birthday, which is there in our database, in our dictionary.

Now, if I have already seen that dictionary, that value before in my dictionary, then this isKey will turn true, and if isKey returns true, then I do not need to update birthdays, because it is already true. But I can update duplicates, because now I know that this value is a duplicate. So, I will create an entry in duplicates, which says that the duplicates of this date of birth is true.

So, in this way, as I go along, each time I see a date of birth, either it will update birthday for the first time, and set it to true, or if birthdays already has a key for this thing, it will create an entry or a reset the entry in duplicates to true. So, in this way, at the end of this all the keys, which are present in duplicates record dates of birth which occur across students.

So, supposing, we want to go one step further and we want to record not just the fact that, there are duplicates. So, I mean, what we now know is from the keys of duplicates we know which dates of birth are common. We know, which dates have more than 1 student on that day, having a birthday on that date, but we do not know, which student's.

So, if you want to know which student's, then we have to keep a little bit more information. So along with the date of birth, we extract from each card the id. Something which uniquely identifies which student is, the sequence number of the card. And now, the update that we saw, when the key is not present, instead of just saying true; we create an entry which is a list. And this list has this first sequence number in that list.

So, the first time we see a date of birth, we will create an entry in birthdays with that key, which is the birthday key. So the key is still the date of birth, but the value now is the list, containing the sequence number of the student who had that birthday. And now, when we do the duplicate check and we find the key already exists, we will of course as before set duplicates to be true to indicate, that we have found, that set that key. So that we have record in duplicate, which dates of birth were duplicate. But we also append this sequence number to the list of sequence numbers associated with this particular birthday.

So now, at the end of this round of processing, duplicates will tell us which keys have multiple students having the same birthday. And then, we look back at birthdays and pick up the same key, the same date of birth. It will tell us the list of students, 2 or more, who have that duplicate birthday. So this gives us a little more information at no extra cost. In terms of processing, we do the same thing except we add, keep track of more information during our iteration.

So, we can as we said before, we can keep list indirectly as dictionary. So this is probably a good place to illustrate that. So instead of storing a list of sequence numbers with each birthday, I am now going to say that for each birthday, I have a dictionary. So when I create this dictionary for the first time, when I create this key instead of assigning it the list sequence number, I will create an empty dictionary and then create a key for that sequence number.

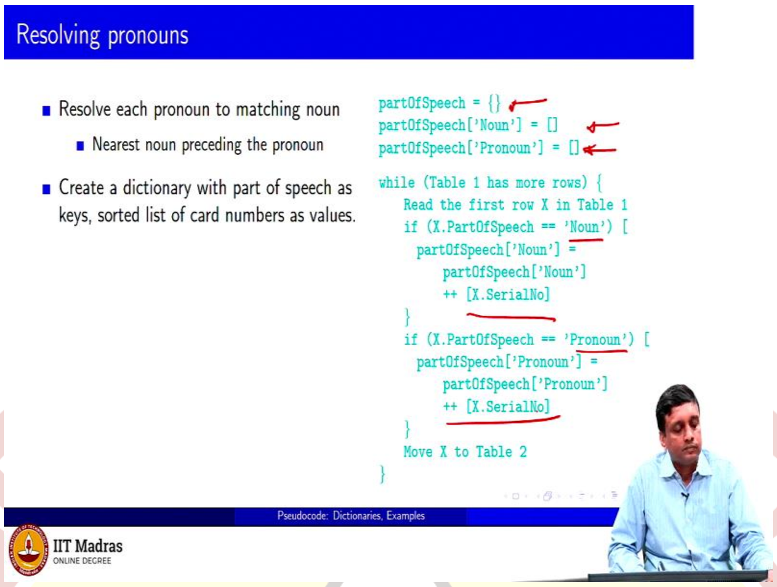
So now, birthdays of a given date of birth is a set of keys. It is not a list of values, but a set of keys, and each key is a sequence number, whose birthday falls on that day. So now, initially I have to create this empty dictionary for that date of birth, and assign it the first key. And when I update it, I create an entry for a new sequence number, which is the duplicate. So, this has no difference in some sense between what we did now and what we did before. The previous case, what we had done was, we had actually recorded this as a list.

So we started off with the list, containing 1 sequence number, and then we appended the sequence number of each duplicate birthday that we saw; instead what we are now saying is that, we keep track of the sequence numbers as keys. So birthdays square bracket dob, is associated with a set of keys; that is, it is a nested dictionary. In this nested dictionary, we have 1 key for every student who has a birthday on that day. So we initialise it with 1 key and then, when we see a duplicate, we create a new key.

So finally again, to know which students are duplicate, we will typically look at this duplicates dictionary. It will tell us, its keys will tell us, which dates of birth have multiple entries. Then we go back to the birthdays dictionary and pick up for that date of birth all the keys. Earlier we picked up the list and we could go through all the values, here we can go through all the keys.

So, it is a kind of different way of keeping track of a list of values. Sometimes it is better, because if I keep up values this way, then I can quickly check, whether a given student is there using isKey. If I have a list of values, there is no way to process it except to go through the list. So when I need to, then quickly check something. Sometimes it is useful to keep this list kind of encoded in this way as a dictionary.

(Refer Slide Time: 10:30)



Resolving pronouns

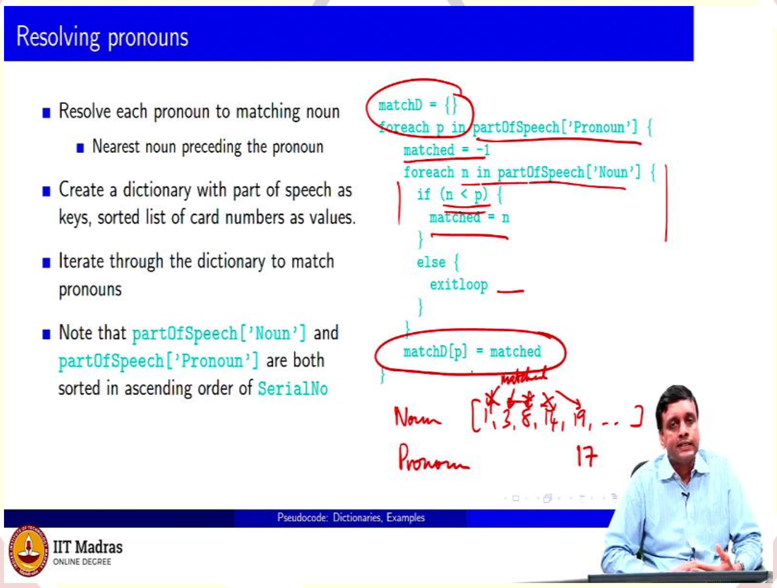
- Resolve each pronoun to matching noun
  - Nearest noun preceding the pronoun
- Create a dictionary with part of speech as keys, sorted list of card numbers as values.

```
partOfSpeech = {}
partOfSpeech['Noun'] = []
partOfSpeech['Pronoun'] = []

while (Table 1 has more rows) {
  Read the first row X in Table 1
  if (X.PartOfSpeech == 'Noun') {
    partOfSpeech['Noun'] =
        partOfSpeech['Noun']
        ++ [X.SerialNo]
  }
  if (X.PartOfSpeech == 'Pronoun') {
    partOfSpeech['Pronoun'] =
        partOfSpeech['Pronoun']
        ++ [X.SerialNo]
  }
  Move X to Table 2
}
```

Pseudocode: Dictionaries, Examples

IIT Madras
ONLINE DEGREE



Resolving pronouns

- Resolve each pronoun to matching noun
  - Nearest noun preceding the pronoun
- Create a dictionary with part of speech as keys, sorted list of card numbers as values.
- Iterate through the dictionary to match pronouns
- Note that partOfSpeech['Noun'] and partOfSpeech['Pronoun'] are both sorted in ascending order of SerialNo

```
matchD = {}
foreach p in partOfSpeech['Pronoun'] {
  matched = -1
  foreach n in partOfSpeech['Noun'] {
    if (n < p) {
      matched = n
    }
    else {
      exitloop
    }
  }
  matchD[p] = matched
}
```

Pseudocode: Dictionaries, Examples

IIT Madras
ONLINE DEGREE

So for the last example, using dictionaries. Let us look at this words database, the paragraph. So we wanted to ask, how we can resolve a pronoun. So when we see a pronoun, we want to know what that pronoun refers to. A pronoun, as you know, is something like he, your, mine, or his, or her; which refers to something which has already been referred to before. If I just start off telling you, he did something, you have no idea who that he is.

So, I will have to first say, Virat Kohli scored 50 runs yesterday, then he got out. Then you know who got out. Well, it is Virat Kohli who got out. So, that is known as resolving the pronoun. We know that, he in the second sentence refers to Virat Kohli in the first sentence.

So, how do we resolve pronouns to nouns? Well, what we did, was a very simplistic thing. We said, we take all the words in position as they occur in our paragraph. We look at the position of a pronoun and well, it must refer to a noun that came before. So, we will go backwards and find the closest noun. So, the closest noun that was before this pronoun, we will decide, is the matching noun. This was just how we did it. Of course in English, it is not always that way, because there may be some other noun in between. But we have to be somewhat, you know, simplistic about this, to make this procedure work.

So, what we do now is we create first a dictionary, which tells us the positions at which each of these relevant parts of speech occur. We need to know where the nouns occur and we need to know where the pronouns occur. So that we can look at a pronoun, where it occurs and find a noun, which occurs closest before it.

So therefore, the natural thing to do is to have a dictionary in which I have a key pronoun, which keeps a list as list of values, which are the positions where there are pronouns. Similarly, I keep a dictionary key, called noun and with this key, I will search a value of all the positions where there are nouns

Now, I can look at every position and pronoun, and look back into the noun thing, and find the smallest, the largest position in noun; which is smaller than the position pronoun. So the first part is just setting up the dictionary. So we start by having a partOfSpeech dictionary, which will create a dictionary entry for every part of speech that we are interested in. Currently, we are interested only in nouns and pronouns. So we initialise those keys and we assign them the empty list. So initially, I have created a key, saying this is the list of positions of nouns; but it is empty. This is the list of positions of pronouns, but it is empty.

And now I do a very simple thing, which is, I go through this table and for every word in the table, I check if the part of speech is a noun, then I append the serial number, that is the position of the current thing to the list of partOfSpeech with the key noun. If the part of speech is a pronoun, then I append this serial number to the list, which contains all the positions of pronouns. So this is a very simple thing. At the end of this, I have got a dictionary, called partOfSpeech, which has 2 keys. The key noun and the key pronoun, and with each key I have associated the list of positions where that particular part of speech occurs.

And now, having got this dictionary, I can now do my pronoun matching. So, I need to match every pronoun. So, I need to go through every position in this list. So, partOfSpeech pronoun is a list of positions in my paragraph, which are pronouns. And I know they are pronouns because I actually calculated them that way. I looked up and added them only if they are pronouns.

Now, for every pronoun in that thing, I need to find out the preceding noun. So the proceeding noun will have a position and the positions in that paragraph start at 0. So, I will assume, by default that the position starts initialise it to minus 1. So matched is going to be the position of the matching noun.

And now, I just do a very simple thing. I go through all the nouns. And anytime I see a noun position which is smaller than the pronoun position, I update matched to n. So, what will happen is, that I will do this repeatedly. So just supposing that I had a pronoun at position 17, and supposing, my noun positions are 1, 3, 8, 14, 19 and so on. So now, this nested loop, what it will do is, that it will say, initially because position 1 is below 17. It will say, matched is, this.

Now, it will move to 3 and say 3 is still less than 17. So it will create, update the value of matched to point to 3, instead of 1. Similarly, it will update it to point to 8 instead of 3, and then 14 instead of 8. And then, it will come to 19, and from 19 onwards, because we have accumulated these positions also in sequence, from 19 onwards everything is going to be bigger than 17, so there will be no further updates.

So when, I come out of this loop, so in fact, when I come out of this loop; I will have 14 as my matched thing. And as a further thing, in case n is not less than p, I can stop looking. Because I know that, these partsOfSpeech were accumulated from beginning to end. So I know that all the positions in the thing are; so once, I see a noun position which is after my pronoun that I am currently enquiring about, there is no need to do anything more. So I can actually exit this foreach.

So, there are couple of extra things going on, which are just optimizations. But the main point is very simple. I am just doing a nested loop. For every pronoun in my pronoun list, I am going through the noun list and finding the last position, which is still before this. And then I am aborting this loop in some sense by doing an exitloop. Then going to the next pronoun, and doing again from the beginning, finding the last noun, that could be the same noun. Sometimes I could use many pronouns, which refer to the same noun, and then only the next pronoun comes. So that is possible.

So, this is how, we do this pronoun matching. And finally, I mean once we have found the matching position, we of course update this dictionary. So, we have this dictionary, called matchD. So matchD is the dictionary of matching positions of nouns. So, this is what we are actually doing this resolving pronouns. So, we need to record for each pronoun, which is the matching noun. So, that is finally done by taking the position that I got in this nested loop, and assigning it to p, the key p in this dictionary matchD.

So, with these 3 examples, one is counting, finding the customer who purchased the maximum number of food items; the second one was to try and look for duplicate entries in our dictionary in order to find out when 2 students had the same birthday; and this one, where we kind of do a list processing, where we create 2 dictionaries with 2 lists, and then we do some nested list processing to resolve pronouns. Hopefully, you have some feel for how dictionaries can be used in coding some of the computational problems that we have done in class.