CHAPTER 8
SEARCHING ALGORITHM

ผศ.ดร.สิลดา อินทรโสธรฉันท์
สาขาวิชาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ มหาวิทยาลัยขอนแก่น

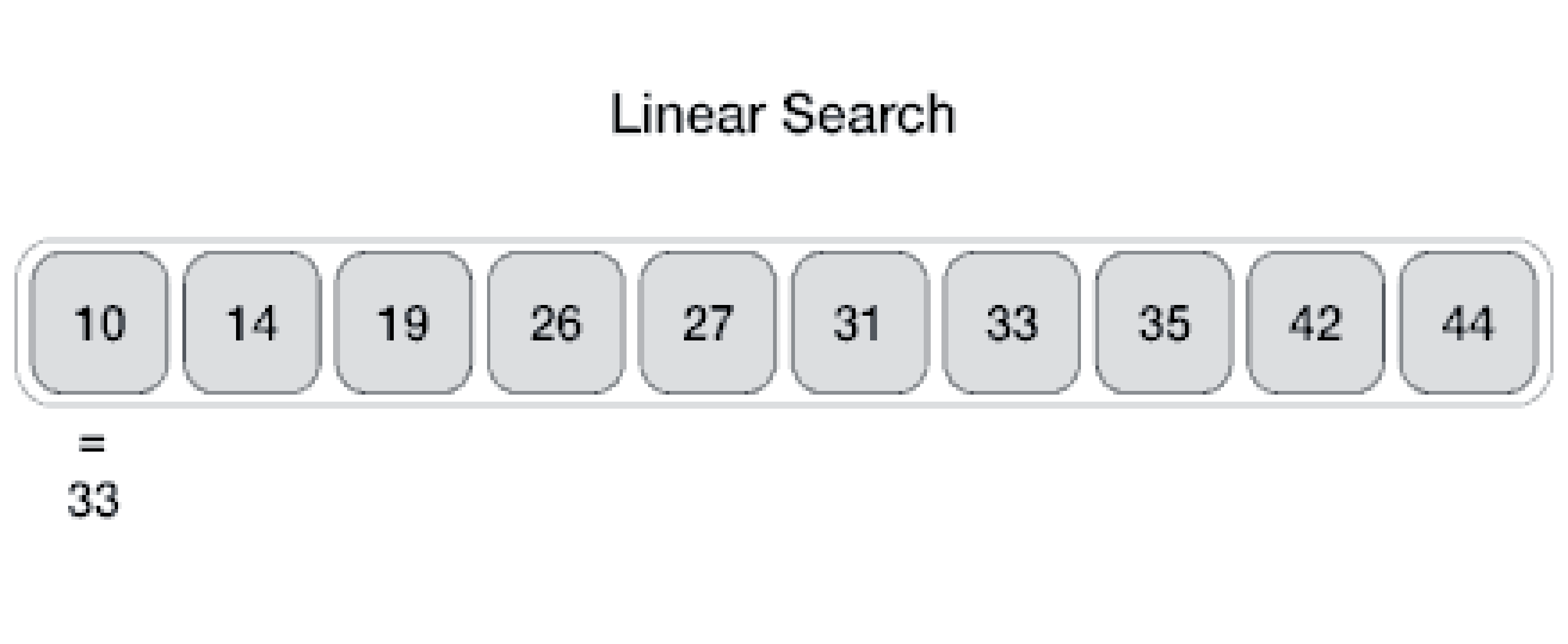
SEARCHING ALGORITHM

- Linear Search
- Binary Search
- Interpolation Search
- Hash Table

LINEAR SEARCH

- Linear search is a very simple search algorithm.
- A sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned.
- Otherwise the search continues till the end of the data collection.

LINEAR SEARCH



- Searching Data = 33

LINEAR SEARCH

```
Linear_Search (list A, value x)
    for each item in the list
        if match item == value
            return the item's location
        end if
    end for
```

BINARY SEARCH

- Binary search is a fast search algorithm with run-time complexity of O(log n).
- This search algorithm works on the principle of divide and conquer.
- This algorithm to work properly, the data collection should be in the sorted form.

BINARY SEARCH

- Binary search looks for a particular item by comparing the middle most item of the collection.
  - If a match occurs, then the index of item is returned.
  - If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
  - Otherwise, the item is searched for in the sub-array to the right of the middle item.
  - This process continues on the sub-array as well until the size of the subarray reduces to zero.

BINARY SEARCH (EXAMPLE 1)

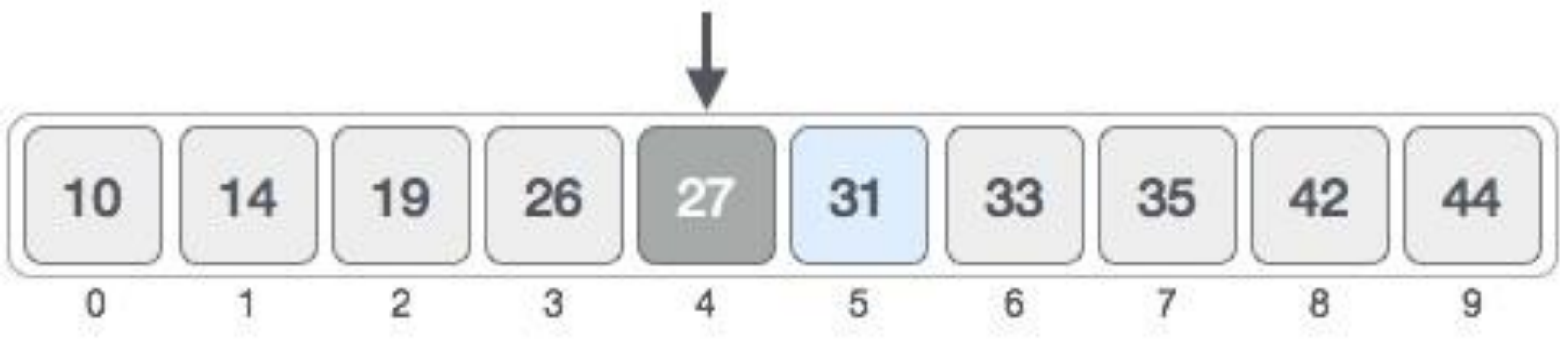
- Assume that we need to search the location of value 31 using binary search.

- First, we shall determine half of the array by using this formula

  mid = low + (high – low)/2

BINARY SEARCH

- **Round 1** : low = 0 , high = 9
- mid = 0 + (9 - 0) / 2 = 4 (integer value of 4.5).
- So, 4 is the mid of the array.



- Compare the value stored at location 4, with the value being searched, i.e. 31.

BINARY SEARCH

- The value at location 4 is 27, which is not a match.
- As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

BINARY SEARCH

- **Round 2 :** Change our low = mid + 1 = 4 + 1 = 5 , high = 9

  mid = 5 + (9 – 5) / 2 = 5 + 2 = 7
- New mid is 7. Compare the value stored at location 7 with our target value 31.

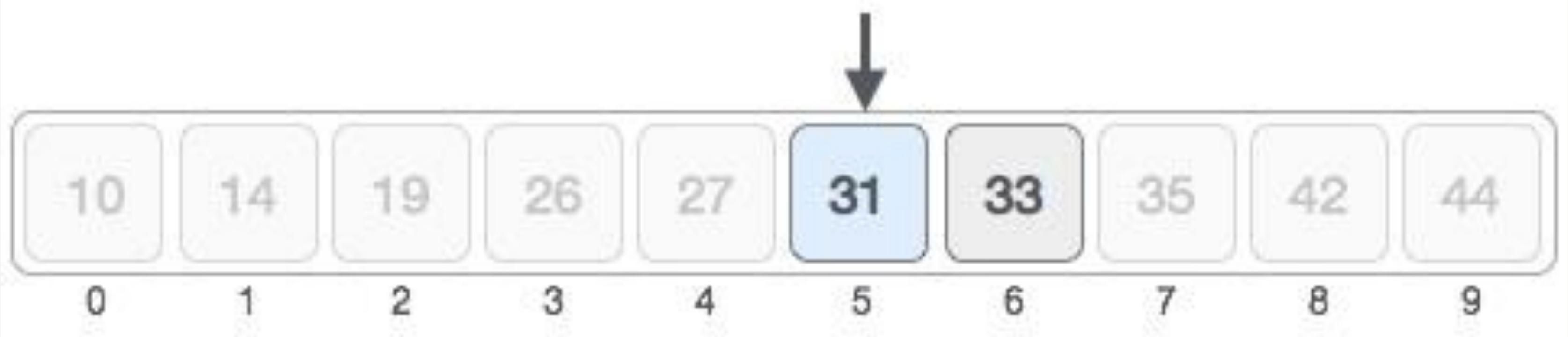
BINARY SEARCH

- The value stored at location 7 is not a match, rather it is more than what we are looking for.
- So, the value must be in the lower part from this location.
- **Round 3** : Change our high = mid - 1 = 7 - 1 = 6 ,   low = 5

  mid = 5 + (6 – 5) / 2 = 5 (integer value of 5.5)

Calculate the mid again. This time it is 5.

BINARY SEARCH



- Compare the value stored at location 5 with our target value.
- It is a match.

BINARY SEARCH

- Conclude that the target value 31 is stored at location 5.
- Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

BINARY SEARCH (EXAMPLE 2)

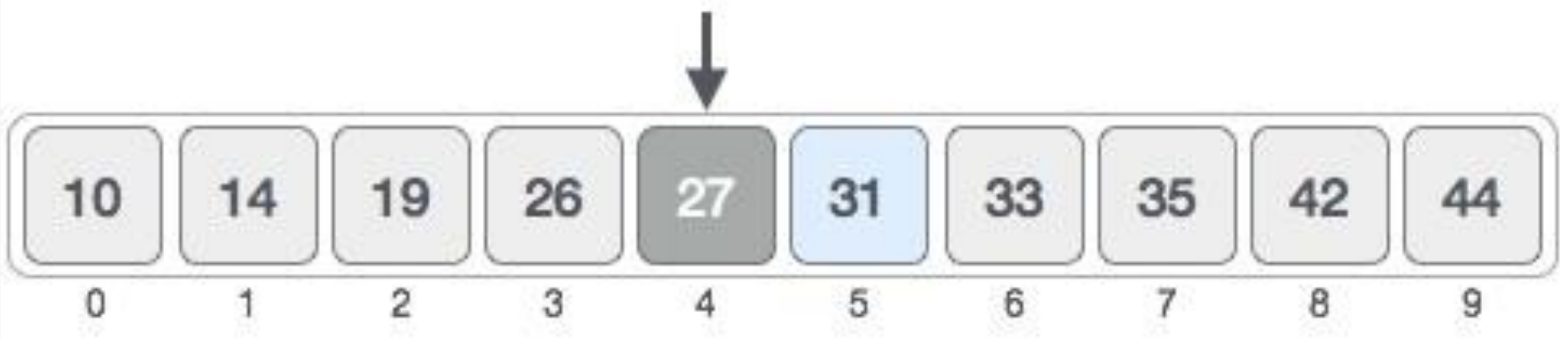
- Assume that we need to search the location of value 32 using binary search.

- First, we shall determine half of the array by using this formula

$$mid = low + (high - low)/2$$

BINARY SEARCH

- **Round 1** : low = 0 , high = 9
- mid = 0 + (9 - 0) / 2 = 4 (integer value of 4.5).
- So, 4 is the mid of the array.



- Compare the value stored at location 4, with the value being searched, i.e. 31.

BINARY SEARCH

- The value at location 4 is 27, which is not a match.
- As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

BINARY SEARCH

- **Round 2** : Change our low = mid + 1 = 4 + 1 = 5 , high = 9

  mid = 5 + (9 – 5) / 2 = 5 + 2 = 7
- New mid is 7. Compare the value stored at location 7 with our target value 32.

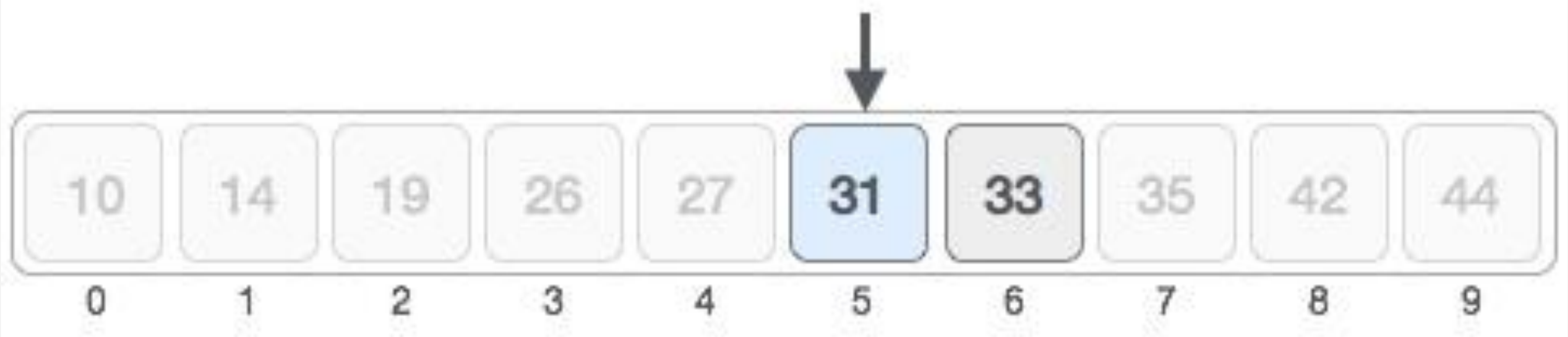
BINARY SEARCH

- The value stored at location 7 is not a match, rather it is more than what we are looking for.
- So, the value must be in the lower part from this location.
- **Round 3** : Change our high = mid - 1= 7 - 1 = 6 ,   low = 5

  mid = 5 + (6 – 5) / 2 = 5 (integer value of 5.5)

Calculate the mid again. This time it is 5.

BINARY SEARCH



- Compare the value stored at location 5 with our target value.
- As the value is greater than 31 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

BINARY SEARCH

- **Round 4** : Change our low = mid + 1 = 5 + 1 = 6 , high = 6

  mid = 6 + (6 – 6) / 2 = 6

- New mid is 6. Compare the value stored at location 6 with our target value 32.

BINARY SEARCH

- The value stored at location 6 is not a match, rather it is more than what we are looking for.
- So, the value must be in the lower part from this location.
- **Round 5** : Change our high = mid - 1= 6 - 1 = 5 , low = 6

  high < low : not found

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1

      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while

end procedure
```

INTERPOLATION SEARCH

- Interpolation search is an improved variant of binary search.
- This search algorithm works on the probing position of the required value.
- For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.

INTERPOLATION SEARCH

- Binary search has a huge advantage of time complexity over linear search.
- Linear search has worst-case complexity of O(n) whereas binary search has O(log n).

INTERPOLATION SEARCH

- Initially, the probe position is the position of the middle most item of the collection.
- If a match occurs, then the index of the item is returned.
- If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item.
- Otherwise, the item is searched in the subarray to the left of the middle item.

INTERPOLATION SEARCH

- This process continues on the sub-array as well until the size of subarray reduces to zero.

```
// when element to be searched is closer to arr[hi]. And
// smaller value when closer to arr[lo]
pos = lo + [ (x-arr[lo])*(hi-lo) / (arr[hi]-arr[Lo]) ]

arr[] ==> Array where elements need to be searched
x     ==> Element to be searched
lo    ==> Starting index in arr[]
hi    ==> Ending index in arr[]
```

INTERPOLATION SEARCH

Step1: In a loop, calculate the value of "pos" using the probe position formula.
Step2: If it is a match, return the index of the item, and exit.
Step3: If the item is less than arr[pos], calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.
Step4: Repeat until a match is found or the sub-array reduces to zero.

INTERPOLATION SEARCH

```
pos = lo + [ (x-arr[lo])*(hi-lo) / (arr[hi]-arr[Lo]) ]

arr[] ==> Array where elements need to be searched
x     ==> Element to be searched
lo    ==> Starting index in arr[]
hi    ==> Ending index in arr[]
```

Example 1

- arr[ ] = [2, 3, 5, 7, 9]
- Target = 7
- lo = 1, hi = 5
- Round 1 :
- Step 1 : pos = 1 + [(7 – 2) * (5 – 1) / (9 – 2)]

  pos = 1 + [5 * 4 / 7] = 1 + 2 = 3

INTERPOLATION SEARCH

Step 2 : compare Target and arr[3] :

7 > 5 : update low index (lo = pos + 1 = 3 + 1 = 4)

lo = 4, hi = 5

Round 2 :

Step 1 : pos = 4 + [(7 – 7) * (5 – 4) / (9 – 7)]

pos = 4 + [0 * 1 / 2] = 4 + 0 = 4

Step 2 : compare Target and arr[4] :

7 = 7 : return pos

INTERPOLATION SEARCH

Example II

- arr[ ] = [2, 3, 5, 7, 9]
- Target = 4
- lo = 1, hi = 5
- Round 1 :

Step 1 : pos = 1 + [(4 – 2) * (5 – 1) / (9 – 2)]

pos = 1 + [2 * 4 / 7] = 1 + 1 = 2

INTERPOLATION SEARCH

Step 2 : compare Target and arr[2] :

4 > 3 : update low index (lo = pos + 1 = 2 + 1 = 3)

lo = 3, hi = 5

Round 2 :

Step 1 : pos = 3 + [(4 – 5) * (5 – 3) / (9 – 5)]

pos = 3 + [-1 * 2 / -4] = 3 + 0 = 3

INTERPOLATION SEARCH

Step 2 : compare Target and arr[3] :

4 < 5 : update high index (hi = pos - 1 = 3 - 1 = 2)

lo = 3, hi = 2

lo > hi : not found

End loop

Return : not found

HASH TABLE

- Hash Table is a data structure which stores data in an associative manner.
- In a hash table, data is stored in an array format, where each data value has its own unique index value.
- Access of data becomes very fast if we know the index of the desired data.

HASH TABLE

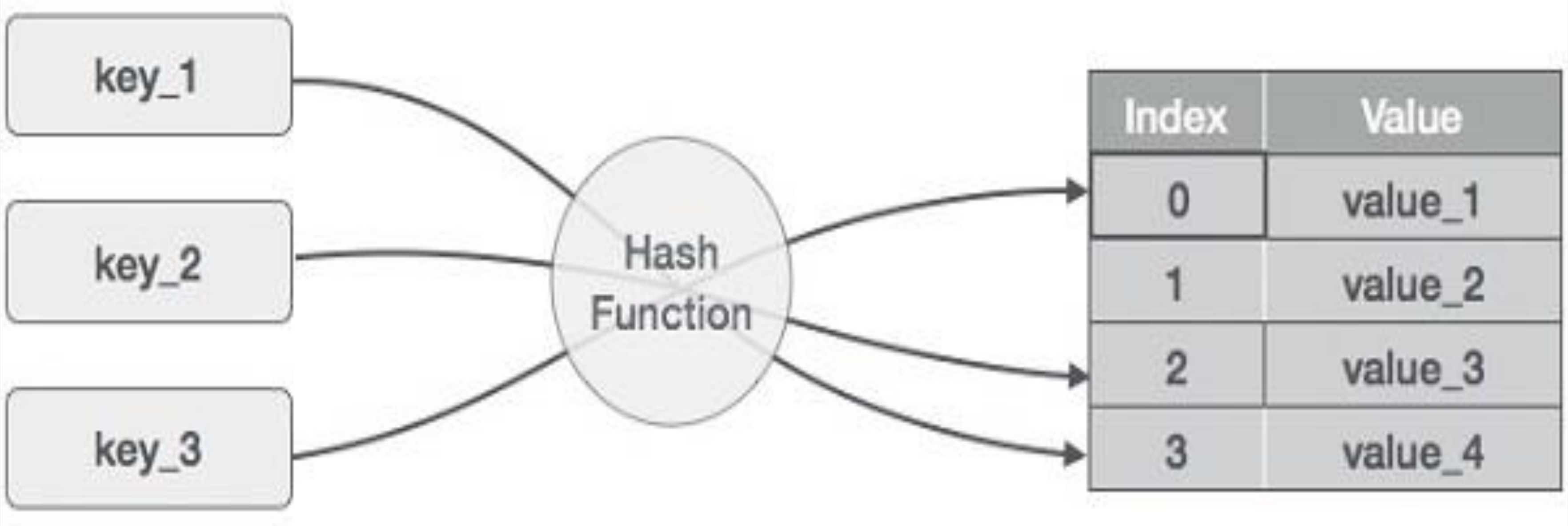
- Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data.
- Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

HASH TABLE

- Hashing is a technique to convert a range of key values into a range of indexes of an array.
- We're going to use modulo operator to get a range of key values.
- Consider an example of hash table of size 20, and the following items are to be stored.
- Item are in the (key, value) format.

HASH TABLE

- (key, value)
- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)



HASH TABLE

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

LINEAR PROBING

- It may happen that the hashing technique is used to create an already used index of the array.
- In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell.
- This technique is called linear probing.

LINEAR PROBING

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18