AP CSA

zhang si 张思

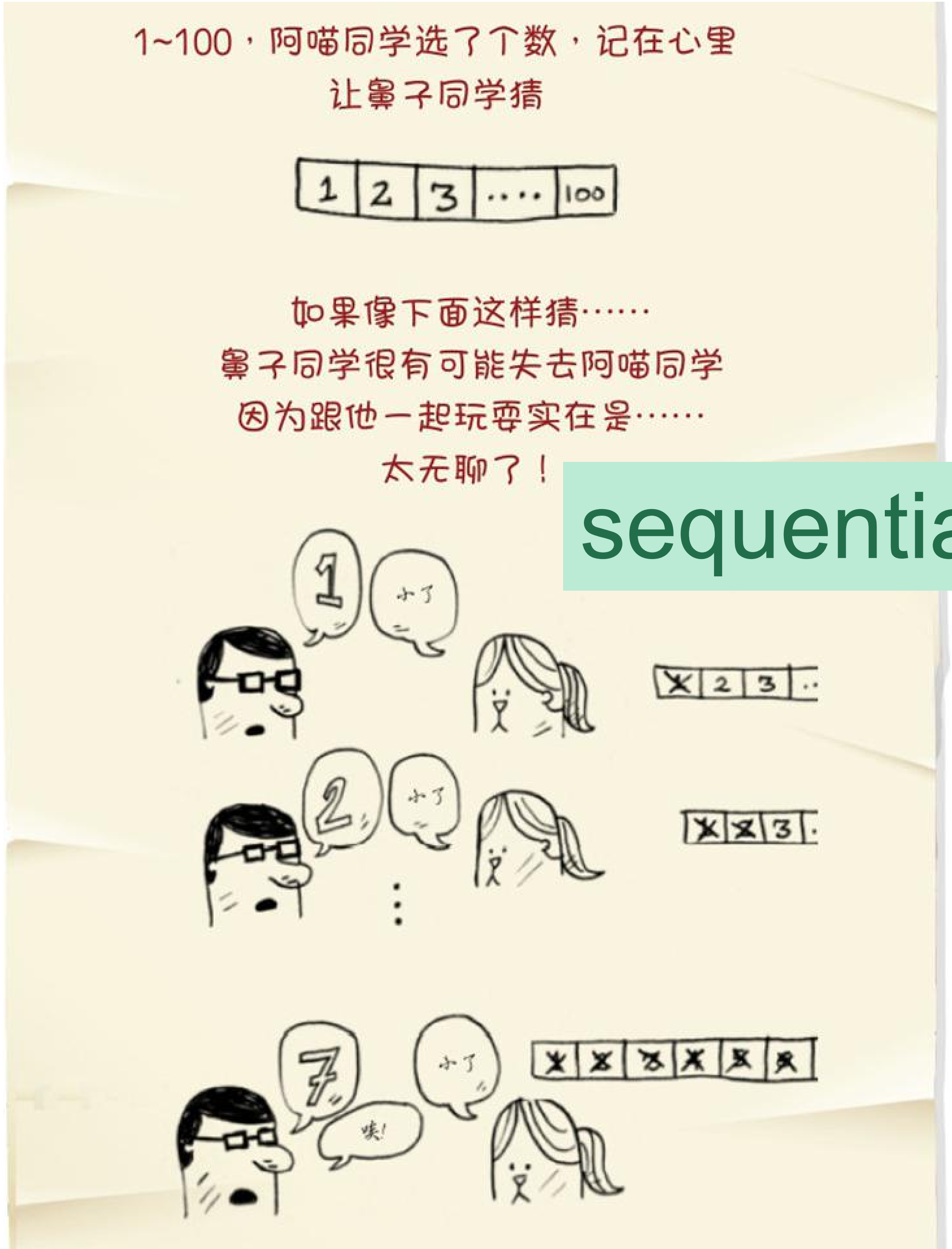
zhangsi@rdfz.cn

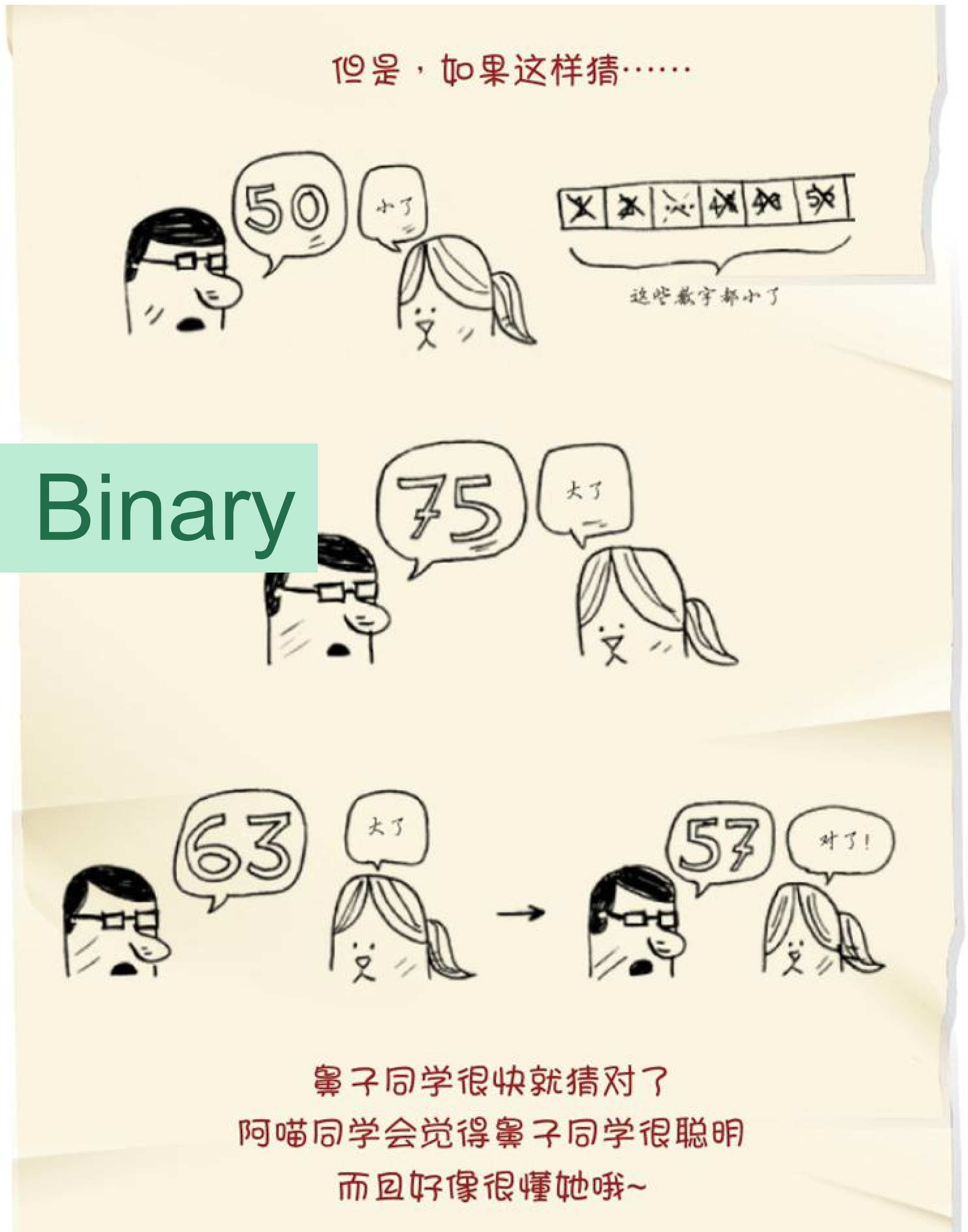
ICC 609

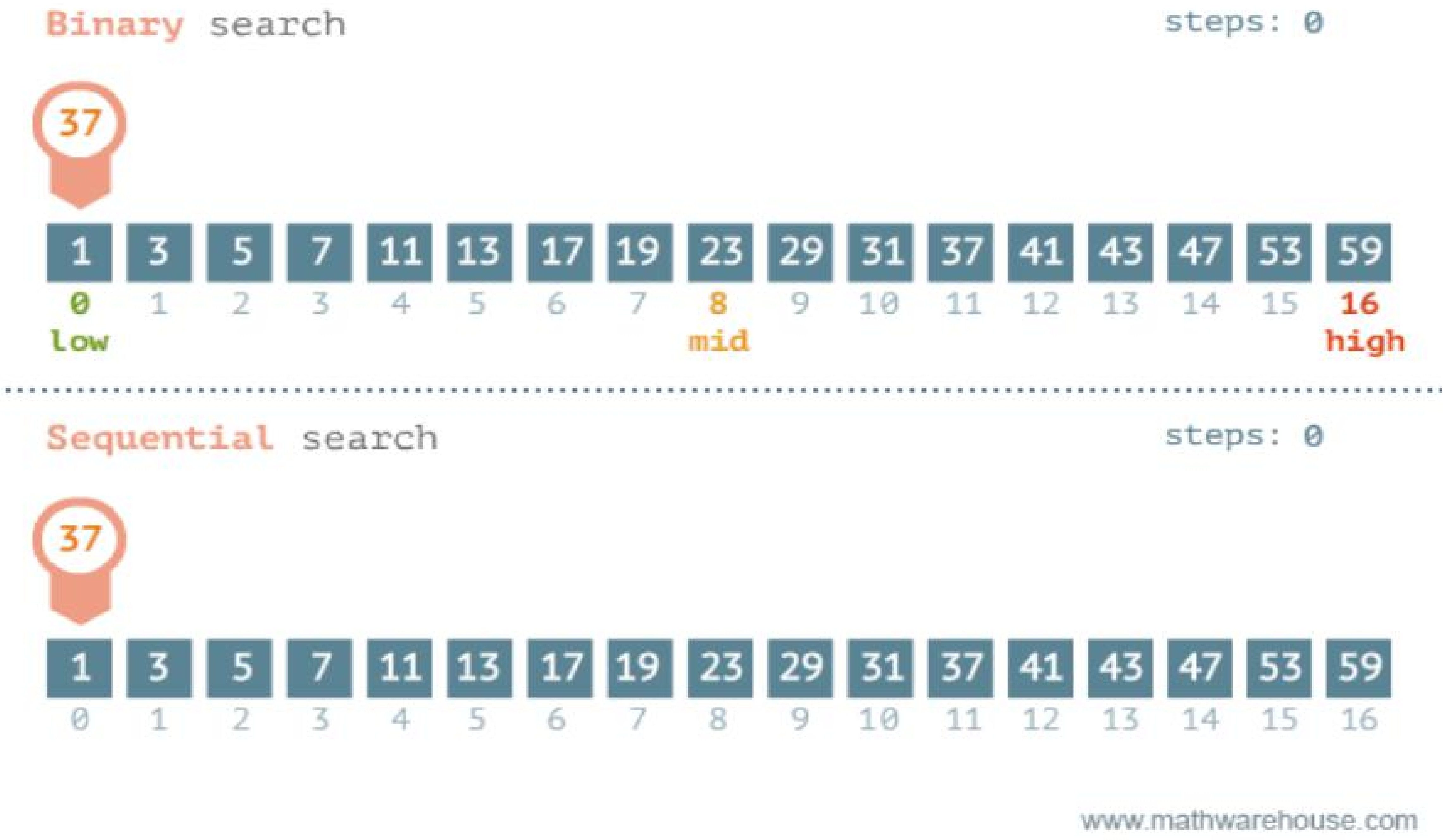
Searching

- https://www.youtube.com/watch?v=DHLCXXX1OtE

sequential vs. Binary

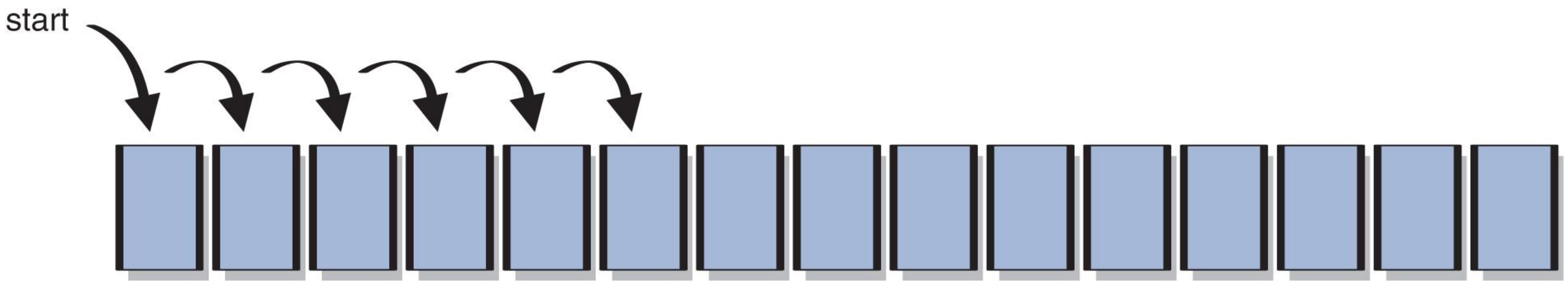





Binary search
steps: 0
37
1 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
low
mid
high
Sequential search
steps: 0
37
1 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
www.mathwarehouse.com

linear or sequential search

- A common task when working with arrays is to search an array for a particular element
- A linear or sequential search examines each element of the array in turn until the desired element is found



- See `Guests.java`

Note:Sequential or linear search is the only method that can be used to find a value in unsorted data.

Searching an array of `int`

```
public int whereIsMyNumber(int magicNumber, int [] myNumbers)
{
    for (int index = 0; index < myNumbers.length; index++)
    {
        if (myNumbers[index] == magicNumber)
        {
            return index;
        }
    }
    return -1;
}
```

Searching an ArrayList of Integer

```
public int where(int magicNumber,ArrayList<Integer> realNumbers)
{
        for(int index=0;index<realNumbers.size();index++)
        {
                if(magicNumber==realNumbers.get(index))
                {
                        return index;
                }
        }
        return -1;
}
```

Searching linear structure

Finding information with a computer is something we need to know how to do. Linear search algorithms are BEST used when we do not have any idea about the order of the data and so we need to look at each element to determine if what we are looking for is in fact inside the array or **`ArrayList`**.

When searching, we do need to remember that different data types require different comparisons!

- When looking at **`int`** values, the == operator is the tool to use!
- When searching for a **`double`** value, we need to make sure the value is close enough by doing some math!
- **`Object`** instances should always use the **`.equals(otherThing)`** method to check for a match!

Searching an ArrayList of Double

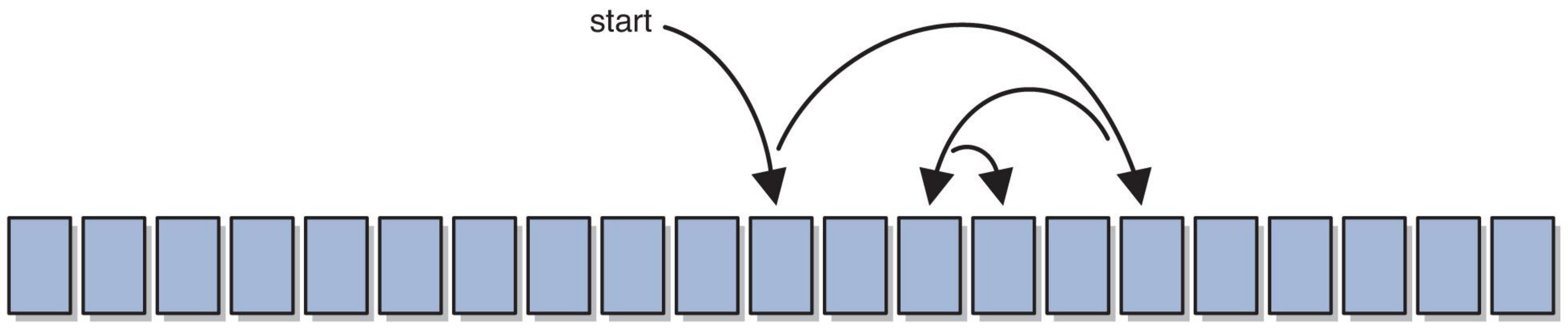
```
public int where(double magicNumber,ArrayList<Double> realNumbers)
{
      for(int index=0;index<realNumbers.size();index++)
      {
            if(Math.abs(magicNumber-realNumbers.get(index))<1e-3)
            {
                  return index;
            }
      }
      return -1;
}
```

Searching an **`ArrayList`** of Book for a **`String`**

```
public int findTheWord(String searchedPhrase, ArrayList<Book> myBooks)
{
    for (int index = 0; index < myBooks.size(); index++)
    {
        Book currentBook = myBooks.get(index);
        String currentPhrase = currentBook.getDescription();
        if (currentPhrase.equals(searchedPhrase))
        {
            return index;
        }
    }
    return -1;
}
```

Binary search

- A binary search is more efficient than a linear search but it can only be performed on an ordered list
- A binary search examines the middle element and moves left if the desired element is less than the middle, and right if the desired element is greater
- This process repeats until the desired element is found



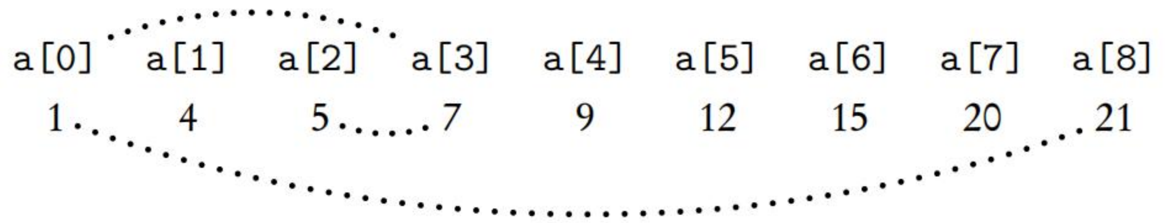
- Implement a binary search algorithm

```java
public int binarySearch(int[] elements, int target) {
     int left = 0;
     int right = elements.length - 1;
     while (left <= right)
     {
        int middle = (left + right) / 2;
        if (target < elements[middle])
        {
           right = middle - 1;
        }
        else if (target > elements[middle])
        {
           left = middle + 1;
        }
        else {
           return middle;
        }
      }
      return -1;
   }
```

NOTE

When `low` and `high` cross, there are no more elements to examine, and `key` is not in the array.

Example: suppose 5 is the key to be found in the following array:



First pass:	`mid = (8+0)/2 = 4.`	Check `a[4]`.
Second pass:	`mid = (0+3)/2 = 1.`	Check `a[1]`.
Third pass:	`mid = (2+3)/2 = 2.`	Check `a[2]`. Yes! Key is found.

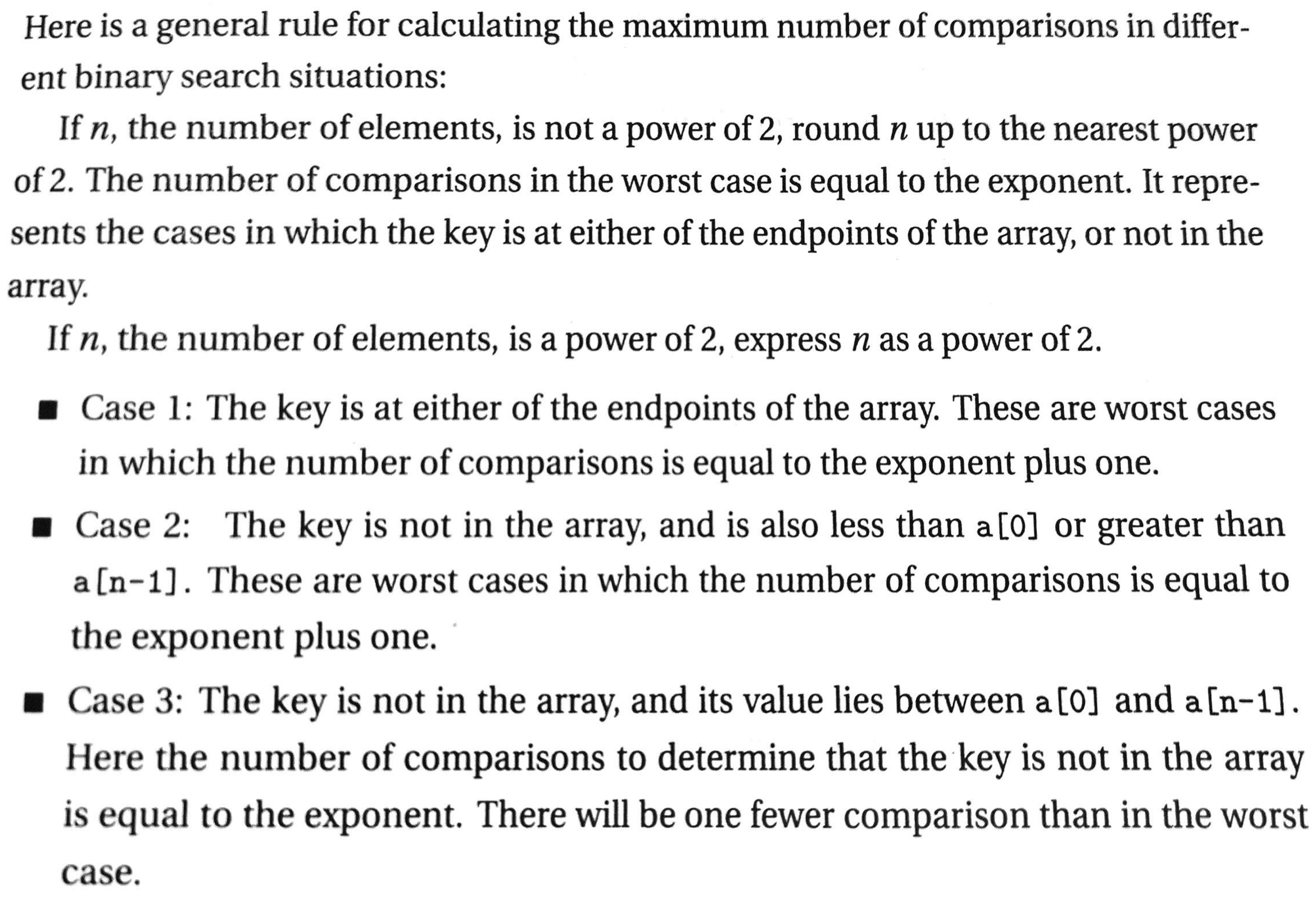
Analysis of Binary Search:

1. In the best case, the key is found on the first try (i.e., `(low + high)/2` is the index of `key`).
2. In the worst case, the key is not in the list or is at either end of a sublist. Here the n elements must be divided by 2 until there is just one element, and then that last element must be tested. An easy way to find the number of comparisons in the worst case is to round n up to the next power of 2 and take the exponent. For example, in the array above, $n = 9$. Suppose 21 were the key. Round 9 up to 16, which equals 2^4. Thus you would need four comparisons to find it. Try it!

Analysis of Binary

- As a simple example, array int[] a={3,7,9,11},
  - if the key is 7, how many comparisons do you need?
  - if the key is 3 or 11(an endpoint of the array),how many comparisons?
  - how about if the key is 1 or 20 (outside the range of values and not in the array)?
  - how about if the key is 8 (not in the array, but inside the range of values)?

Analysis of Binary Search



Here is a general rule for calculating the maximum number of comparisons in different binary search situations:

If *n*, the number of elements, is not a power of 2, round *n* up to the nearest power of 2. The number of comparisons in the worst case is equal to the exponent. It represents the cases in which the key is at either of the endpoints of the array, or not in the array.

If *n*, the number of elements, is a power of 2, express *n* as a power of 2.

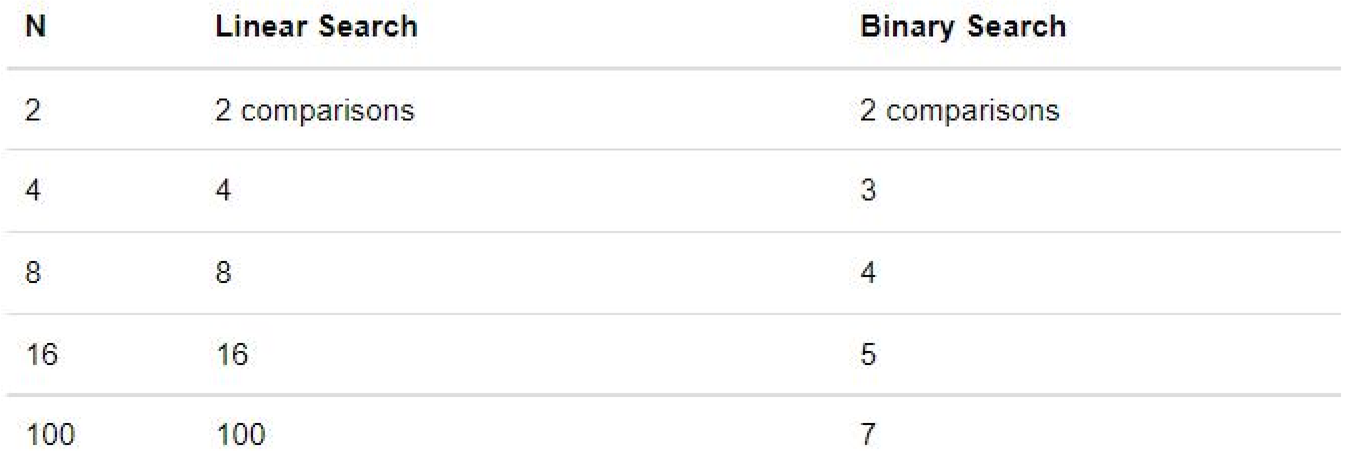
- Case 1: The key is at either of the endpoints of the array. These are worst cases in which the number of comparisons is equal to the exponent plus one.
- Case 2: The key is not in the array, and is also less than `a[0]` or greater than `a[n-1]`. These are worst cases in which the number of comparisons is equal to the exponent plus one.
- Case 3: The key is not in the array, and its value lies between `a[0]` and `a[n-1]`. Here the number of comparisons to determine that the key is not in the array is equal to the exponent. There will be one fewer comparison than in the worst case.

Run times

- How do we choose between two algorithms that solve the same problem?
- They usually have different characteristics and runtimes which measures how fast they run. For the searching problem, it depends on your data.
- Binary search is much faster than linear search, especially on large data sets, but it can only be used on sorted data.
- Often with runtimes, computer scientist think about the worst case behavior.
- With searching, the worst case is usually if you cannot find the item.
- With linear search, you would have to go through the whole array before realizing that it is not there, but binary search is much faster even in this case because it eliminates half the data set in each step.
- We can measure an informal runtime by just counting the number of steps.

Run times

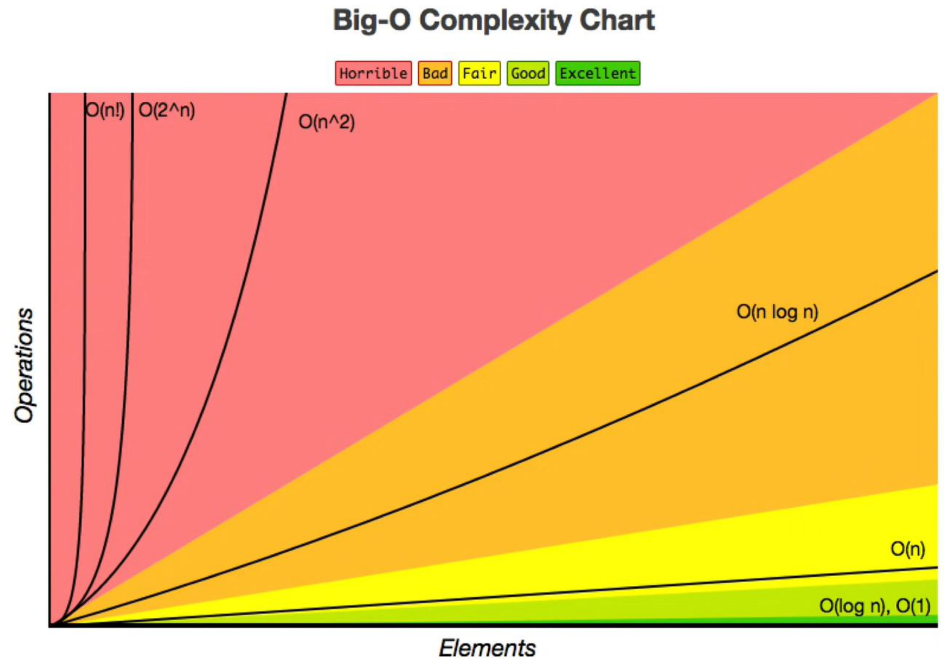
- Here is a table that compares the worst case runtime of each search algorithm given an array of n elements.
- The runtime here is measured as the number of times the loop runs in each algorithm or the number of elements we need to check in the worst case when we don’ t find the item we are looking for.
- Notice that with linear search, the worst case runtime is the size of the array n, because it has to look through the whole array. For the binary search runtime, we can calculate the number of times you can divide n in half until you get to 1.



N	Linear Search	Binary Search
2	2 comparisons	2 comparisons
4	4	3
8	8	4
16	16	5
100	100	7

big O notation

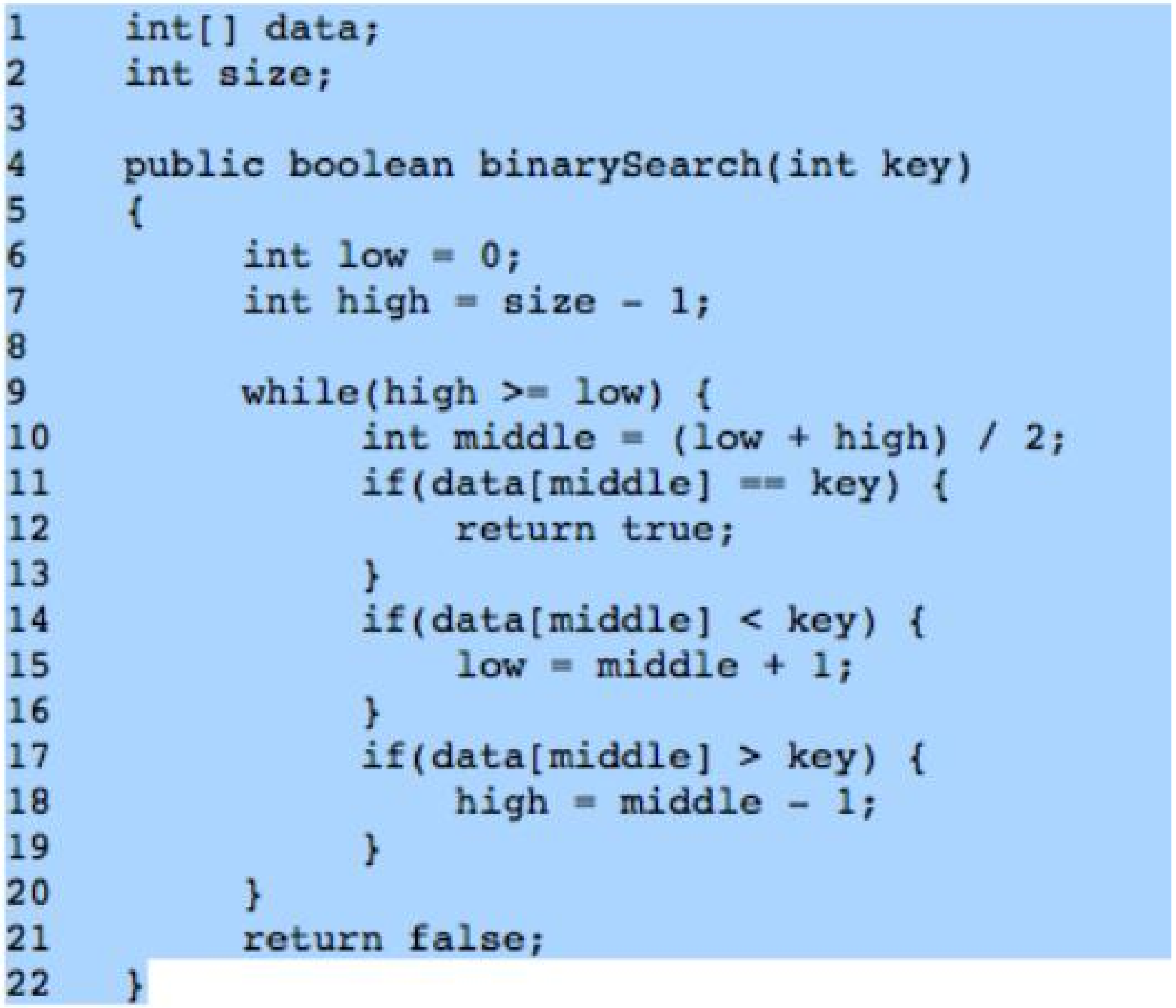
- Runtimes can be described with mathematical functions. For an array of size n, linear search runtime is a linear function, and binary search runtime is a function of log base 2 of n (or $\log_2 n + 1$ comparisons). This is called the big-O runtime function in computer science, for example O(log n) vs. O(n). You can compare the growth of functions like n and log2n as n, the data size, grows and see that binary search runs much faster for any n.



Source: bigocheatsheet.com

Essential knowledge

- There are standard algorithms for searching.
    - linear search
    - binary search
- Sequential/linear search algorithms check each element in order until the desired value is found or all elements in the array or ArrayList have been checked.



```java
int[] data;
int size;

public boolean binarySearch(int key)
{
     int low = 0;
     int high = size - 1;

     while(high >= low) {
          int middle = (low + high) / 2;
          if(data[middle] == key) {
              return true;
          }
          if(data[middle] < key) {
              low = middle + 1;
          }
          if(data[middle] > key) {
              high = middle - 1;
          }
     }
     return false;
}
```

Sorting

- There are many sorting algorithms to put an array or ArrayList elements in alphabetic or numerical order.
- The three sorting algorithms that you need to know for the AP CS A exam are:
  - **Selection Sort** - Select the smallest item from the current location on to the end of the array and swap it with the value at the current position. Do this from index 0 to the array length - 2. You don’t have to process the last element in the array, it will already be sorted when you compare the prior element to the last element.
  - **Insertion Sort** - Insert the next unsorted element in the already sorted part of the array by moving larger values to the right. Start at index 1 and loop through the entire array.
  - **Merge sort** - Break the elements into two parts and recursively sort each part. An array of one item is sorted (base case). Then merge the two sorted arrays into one. MergeSort will be covered in Unit 10.

Selection sort

- https://www.bilibili.com/video/BV1wy4y1J7Ki?from=search&seid=4110996557778018806&spm_id_from=333.337.0.0
- https://www.youtube.com/watch?v=g-PGLbMth_g&t=144s
- folk dance:
- https://www.youtube.com/watch?v=Ns4TPTC8whw&feature=emb_logo

Selection sort

- The approach of Selection Sort:
  - select a value and put it in its final place into the list
  - repeat for all other values
- In more detail:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places

Selection sort

- An example:

```
original:          3    9    6    1    2
smallest is 1:     1    9    6    3    2
smallest is 2:     1    2    6    3    9
smallest is 3:     1    2    3    6    9
smallest is 6:     1    2    3    6    9
```

- See `SortGrades.java`
- See `Sorts.java` -- the `selectionSort` method

Swapping

- *Swapping is the process of exchanging two values*

- Swapping requires three assignment statements

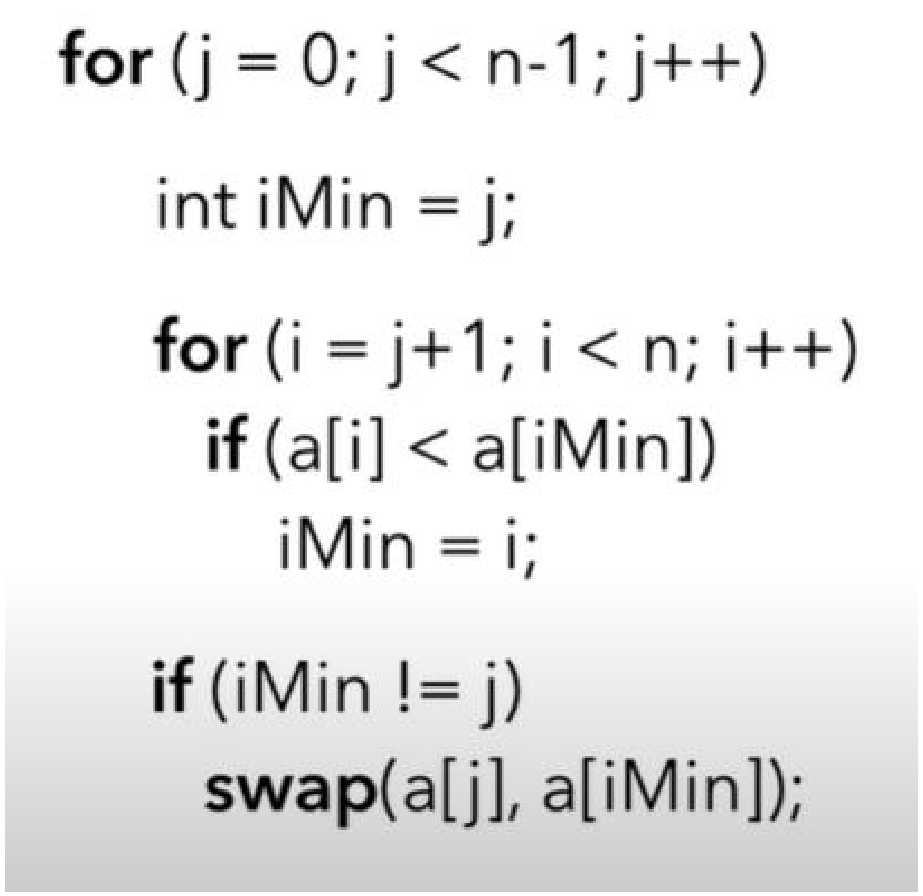
```
temp = first;

first = second;

second = temp;
```

Selection sort

- pseudocode



```
for (j = 0; j < n-1; j++)
    int iMin = j;
    for (i = j+1; i < n; i++)
      if (a[i] < a[iMin])
        iMin = i;
    if (iMin != j)
      swap(a[j], a[iMin]);
```

java code

```
public static void selectionSort(int[] elements)
{
  for (int j = 0; j < elements.length – 1; j++)
  {
     int minIndex = j;
     for (int i = j+1; i < elements.length; i++)
     {
        if (elements[i] < elements[minIndex])
        {
           minIndex = i;
        }
     }
     if(minIndex!=j)
     {
         int temp = elements[j];
         elements[j] = elements[minIndex];
         elements[minIndex] = temp;
     }
   }
}
```

Selection sort

- selection sort in ArrayList

```
public static void selectionSort(ArrayList<Integer> elements)
{
  for (int j = 0; j < elements.size()-1; j++)
  {
     int minIndex = j;
     for (int i = j+1; i < elements.size(); i++)
     {
        if (elements.get(i).compareTo(elements.get(minIndex))<0)
        {
           minIndex = i;
        }
     }
     if(minIndex!=j)
     {
         Integer temp = elements.get(j);
         elements.set(j,elements.get(minIndex));
         elements.set(minIndex, temp);
     }
   }
}
```

Selection sort——Questions

Can an enhanced for loop be used?

No, The Selection Sort algorithm needs to know the index of the items it is working with.

How does the swap occur?

A third variable is needed to temporarily hold on to the swapped value from the array since variables can only hold one thing at time.

Insertion sort

- https://www.youtube.com/watch?v=JU767SDMDvA
- folk dance:
- https://www.youtube.com/watch?v=ROalU379l3U&t=242s

Insertion sort

- The approach of Insertion Sort:
  - pick any item and insert it into its proper place in a sorted sublist
  - repeat until all items have been inserted
- In more detail:
  - consider the first item to be a sorted sublist (of one item)
  - insert the second item into the sorted sublist, shifting the first item as needed to make room to insert the new addition
  - insert the third item into the sorted sublist (of two items), shifting items as necessary
  - repeat until all values are inserted into their proper positions

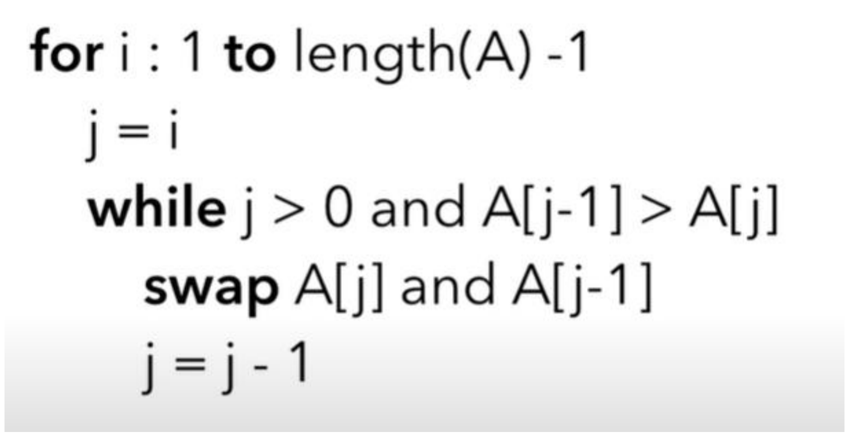
Insertion Sort

- An example:

```
original:    3   9   6   1   2
insert 9:    3   9   6   1   2
insert 6:    3   6   9   1   2
insert 1:    1   3   6   9   2
insert 2:    1   2   3   6   9
```

- See `Sorts.java` -- the `insertionSort` method

Insertion sort



```
for i : 1 to length(A) -1
  j = i
  while j > 0 and A[j-1] > A[j]
    swap A[j] and A[j-1]
    j = j - 1
```

```
public static void insertionSort(int[] elements)
  {
     for (int i = 1; i < elements.length; i++)
     {
        int temp = elements[i];
        int possibleIndex = i;
        while (possibleIndex > 0 && temp < elements[possibleIndex - 1])
        {
           elements[possibleIndex] = elements[possibleIndex - 1];
           possibleIndex--;
        }
        elements[possibleIndex] = temp;
     }
  }
```

```
for (i=1;i<n-1,i++)
   temp=a[i]
    j=i;
   while(j>0 and a[j-1]>a[j])
        a[j]=a[j-1]
         j--
    a[j]=temp
```

Insertion sort

- insertion sort in ArrayList

```
public static void insertionSort(ArrayList<Integer> elements)
   {
      for (int i = 1; i < elements.size(); i++)
      {
         Integer temp = elements.get(i);
         int possibleIndex = i;
         while (possibleIndex > 0 && temp.compareTo(elements.get(possibleIndex - 1))<0)
         {
            elements[possibleIndex] = elements[possibleIndex - 1];
            possibleIndex--;
         }
         elements.set(possibleIndex,temp);
      }
  }
```

Insertion sort——Questions

Why use a `while` loop as the inner loop?

As soon as the condition is failed a while loop will not execute again, if a for loop was chosen a much more complex middle section of the for loop header is required.

Comparing Sorts

- Time efficiency refers to how long it takes an algorithm to run
- Space efficiency refers to the amount of space an algorithm uses
- Algorithms are compared to each other by expressing their efficiency in *big-O notation*
- An efficiency of O(n) is better than O(n^2), where n refers to the size of the input
- Time efficiency O(2^n) means that as the size of the input increases, the running time increases exponentially

Comparing Sorts

- Both Selection and Insertion sorts are similar in efficiency
- They both have outer loops that scan all elements, and inner loops that compare the value of the outer loop with almost all values in the list
- Approximately n^2 number of comparisons are made to sort a list of size n
- We therefore say that these sorts have efficiency $O(n^2)$, or are of order n^2
- Other sorts are more efficient: $O(n \log_2 n)$

Essential knowledge

- Selection sort and insertion sort are iterative sorting algorithms that can be used to sort elements in an array or ArrayList.
- Informal run-time comparisons of program code segments can be made using statement execution counts.