Data Structures and Algorithms

XMUT-COMP 103 - 2026 T1

Search Algorithms

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

Topics

- Intro to search
- Ways to search.
- Ways to rate searching algorithms.
- Searching with sorted list.
- Binary search (recursive).
- Binary search (iterative).
- Cost of binary search.
- Another form of binary searching.

Search Algorithm

Searching is a process of locating a specific element or item within a collection of data, such as arrays, lists, trees, or other structured representations.

The search objectives are typically to determine whether the desired element exists in the data and, if so, to identify its precise location or retrieve it.

Why searching is important:

- **Efficiency:** Efficient searching algorithms improve program performance.
- **Data retrieval:** Quickly find and retrieve specific data from large datasets.
- **Database systems:** Enable fast querying of databases.
- **Problem solving:** Used in a wide range of problem-solving tasks.

Ways to Rate Search Algorithm

Target Element: A specific target element or item to find in the data collection, i.e. a value, a record, a key, or any other data entity of interest.

Search Space: The entire collection of data for finding the target element. Depending on the data structure used, the search space may vary in size and organisation.

Complexity: Different levels of complexity depending on the data structure and the algorithm used, i.e. often measured in terms of time and space requirements.

Deterministic vs Non-deterministic: For a comparison, binary search is deterministic, i.e., follows a clear and systematic approach.
Others, such as linear search, are non-deterministic, i.e. need to examine the entire search space in the worst case.

Example of Search Algorithm

Some examples of popular basic searching algorithms

- Linear search
- Binary search
- Ternary search
- Jump search
- Interpolation search
- Fibonacci search
- Exponential search

Search Algorithm

Essential tools in computer science for locating specific items within a data collection.

We will mainly focus on searching for an item in an array. There are **two** most common algorithms, depending on the type of input array.

Linear Search: It is used for an *unsorted array*.

- It mainly does a one-by-one comparison of the item to be searched with the array elements.
- It takes linear or O(n) Time.

Binary Search: It is used for *a sorted array*.

- It mainly compares the array's middle element first, and if the middle element is the same as the input, then it returns.
- Otherwise, it searches either the left half or the right half based on the comparison result (whether the middle element is smaller or greater).
- This algorithm is faster than linear search and takes O(Log n) time.

Linear Search

O(n) Time and O(1) Space
It is the simplest searching algorithm that checks each element sequentially until the key is found or the collection is fully traversed. Works on both sorted and unsorted data

How does it work?

Iterate over **all** the elements of the array
Check whether the current element equals the target element.
If we find any element to be equal to the target element, then return the index of the current element. Otherwise, if no element is equal to the target element, then return -1 as the element is not found.

Linear search is also known as **sequential search**.

Linear Search

Let's say we have an array arr[] and searching for x

```
class LinearSearch {
    public static int search(int arr[], int N, int x)
    {
        // Iterate over the array in order to find the key x
        for (int i = 0; i < N; i++) {
            if (arr[i] == x)
                return i;
        }
        return -1;
    }

    public static void main(String args[]) {
     ….
    }
}
```

```
public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;

        int result = search(arr, arr.length, x);
        if (result == -1)
            System.out.print(
                "Element is not present in array");
        else
            System.out.print("Element is present at index "
                                + result);
    }
```

Linear Search

```
class LinearSearch {
    public static int search(int arr[], int N, int x)
    {…}

    public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;

        int result = search(arr, arr.length, x);
        if (result == -1)
            UI.println("Element is not present in array");
        else
            UI.println("Element is present at index " + result);
    }
}
```

Linear Search

Time Complexity:

•**Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)

•**Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.

•**Average Case:** O(N)

Applications of Linear Search Algorithm:

•**Unsorted Lists:** When we have an unsorted array or list, linear search is most commonly used to find any element in the collection.

•**Small Data Sets:** Linear Search is preferred over binary search when we have small data sets with

•**Searching Linked Lists:** In linked list implementations, linear search is commonly used to find elements within the list. Each node is checked sequentially until the desired element is found.

•**Simple Implementation:** Linear Search is much easier to understand and implement as compared to Binary Search or Ternary Search.

Linear Search

Time Complexity:

•**Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)

•**Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.

•**Average Case:** O(N)

Applications of Linear Search Algorithm:

•**Unsorted Lists:** When we have an unsorted array or list, linear search is most commonly used to find any element in the collection.

•**Small Data Sets:** Linear Search is preferred over binary search when we have small data sets with

•**Searching Linked Lists:** In linked list implementations, linear search is commonly used to find elements within the list. Each node is checked sequentially until the desired element is found.

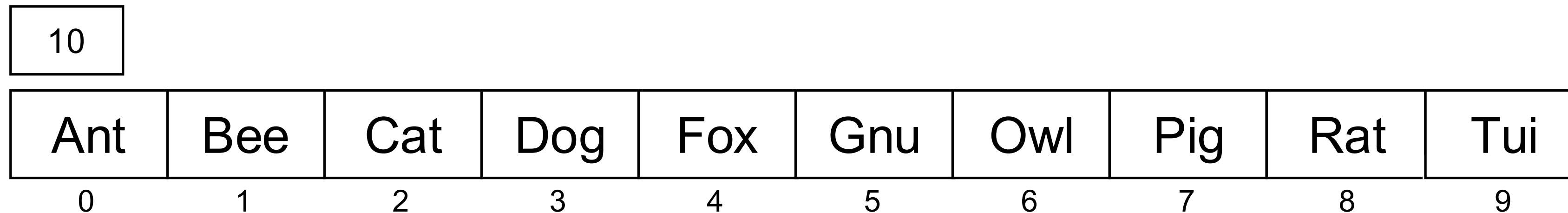
•**Simple Implementation:** Linear Search is much easier to understand and implement as compared to Binary Search or Ternary Search.

Binary Search

Searching for Items in a sorted List.

- Searching for an item in a List is normally O(n)   (contains, indexOf)
- If the List is sorted, we can do much better.

- Binary Search: Finding "Gnu"



  - Look in the middle:
    - if item is middle item   ⇒ return
    - if item is before middle item   ⇒ look in left half
    - if item is after middle item   ⇒ look in right half

Binary Search

Example:

We have arr [] = {2, 5, 8 13, 16, 18, 23, 24,29, 31}

X = 23

23

0	1	2	3	4	5	6	7	8	9
2	5	8	13	16	18	23	24	29	31

Low =0 Mid = 4 High =9

Step 1
Find the mid (4) = 16
16 is less than the key=23,
move to the right

23

0	1	2	3	4	5	6	7	8	9
2	5	8	13	16	18	23	24	29	31

Low =5 Mid = 7 High =9

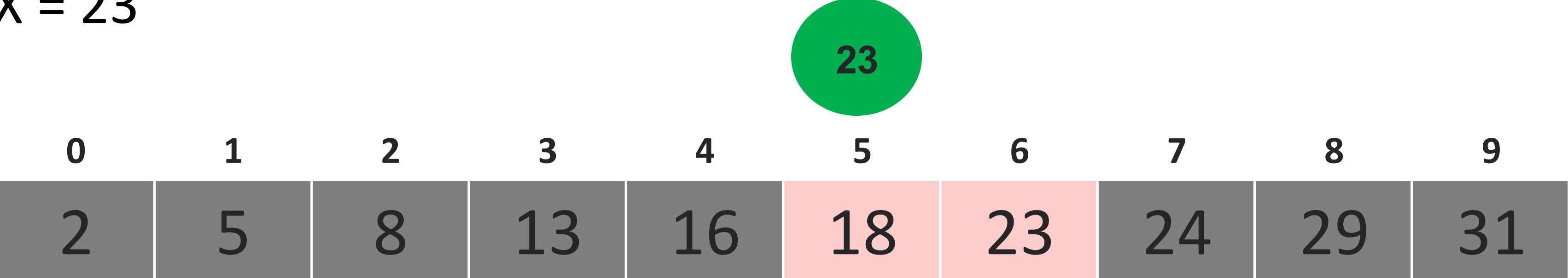
Step 2
Find the mid (7) = 24
24 is greater than key=23,
move to the left

Binary Search

Example:

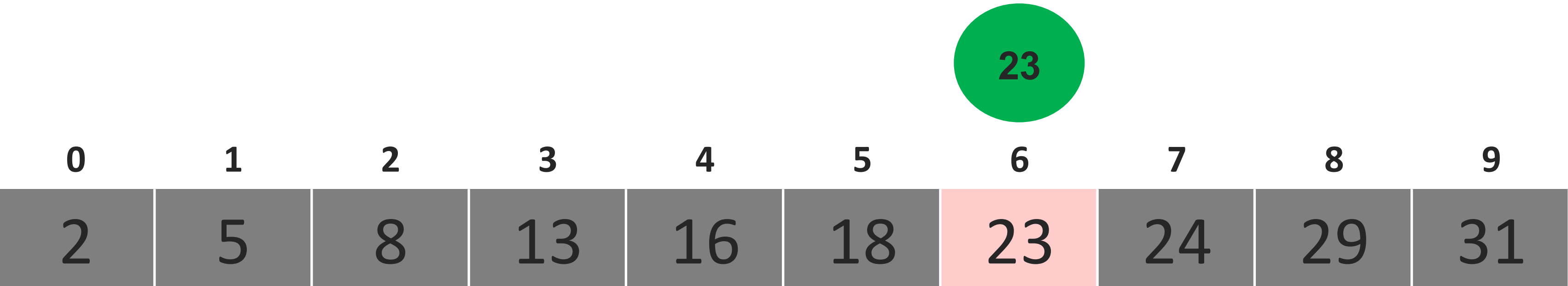
We have arr [] = {2, 5, 8 13, 16, 18, 23, 24,29, 31}

X = 23



Low = 5 High = 6
Mid = 5

Step 3
Find the mid (5) = 18
18 is less than the key=23,
move to the right



Step 4
Find the mid (6) = 23
23 is equal to key=23,
Found the item!

Binary Search

- Operates on a sorted or monotonic search space, repeatedly dividing it into halves to find a target value or optimal answer in logarithmic time **O(log N)**.

How to Implement Binary Search?

It can be implemented in the following two ways

- Iterative Binary Search Algorithm
- Recursive Binary Search Algorithm

Binary Search (recursive)

```
public int indexOf(String value, List<String> data){
    return indexOf(value, data, 0, data.size());
}

public int indexOf(String value, List<String> data, int low, int high){
    // value in [low .. high)  (if present)
    if (low >= high){  return -1;  }            //  value not present
    int mid  =  (low + high) / 2;
    int comp =  value.compareTo(data.get(mid));
    if (comp == 0)        { return mid;   }    // item is present
    else if (comp < 0)   { return indexOf(value, data, low, mid); }      // item in [low .. mid)
    else                       { return indexOf(value, data, mid+1, high);}   // item in [mid+1 .. high)
}
```

Binary Search (recursive)

Cost:

- each recursive call cuts the range in half.
- number of recursive calls = number of times can cut n items in half = $\log_2(n)$
- cost of each line (except recursive calls) = O(1)
- Total cost = O(log(n))

```
public int indexOf(String value, List<String> data, int low, int high){
    // value in [low .. high)  (if present)
    if (low >= high){  return -1;  }            //  value not present
    int mid  =  (low + high) / 2;
    int comp =  value.compareTo(data.get(mid));
    if (comp == 0)        { return mid;   }    // item is present
    else if (comp < 0)   { return indexOf(value, data, low, mid); }       // item in [low .. mid)
    else                       { return indexOf(value, data, mid+1, high);}   // item in [mid+1 .. high)
}
```

Binary Search (iterative)

```
private int indexOf(String value, List<String> data){
    int low = 0;
    int high = data.size();
    // item in [low .. high)  (if present)
    while (low < high){
        int mid  =  (low + high) / 2;
        int comp =  value.compareTo(data.get(mid));
        if (comp == 0) // item is at mid
            return mid;
        if (comp < 0)    // item in [low .. mid)
            high = mid;             // item in [low .. high)
        else   // item in [mid+1 .. high)
            low = mid + 1;          // item in [low .. high)
    }
    return -1;              // item in [low .. high) and low >= high,
}                           // therefore item not present
```

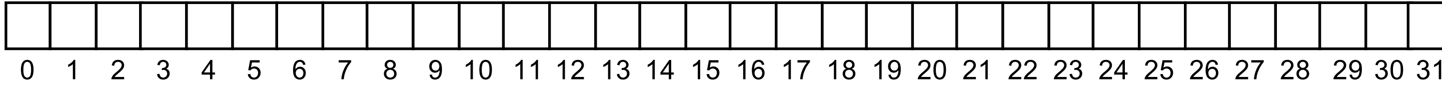
Another form of Binary Search

```
/* Return the index of where the item ought to be, whether present or not.  (!) */
    private int findIndex(String value, List<String> data){
      int low = 0;
      int high  =  data.size();          // index in [low .. high]
      while (low < high){
        int mid  =  (low + high) / 2;
        if (value.compareTo(data.get(mid)) > 0)        // index in [mid+1 .. high]
            low = mid + 1;               // index in [low .. high]  low <= high
        else                      // index in [low .. mid]
            high = mid;                  // index in [low .. high], low<=high
      }
      return low;                 // index in [low .. high] and low = high
                                  // therefore index = low
    }
```

Note: correct position might be at end (index =size)

Binary Search: Cost

- What is the cost of searching if *n* items in set?
  - key step = ?



- Iteration Size of range Cost of iteration

Iteration	Size of range	Cost of iteration
1	n	
2		
k	1	

Example Binary Search

Dictionary Look up

"apple", "antelope","banana", "bear", "cherry", "cat", "date", "deer", "elderberry", "elephant","fig", "fox","grape", "giraffe","honeydew", "horse", "ice apple", "iguana","jackfruit", "jaguar","kiwi", "kangaroo","lemon", "lion","mango", "monkey", "nectarine", "newt", "orange", "ostrich", "papaya", "panda", "quince", "quail", "raspberry", "rabbit", "strawberry", "snake", "tomato", "tiger","ugli fruit", "urial","vanilla", "vulture", "watermelon", "wolf", "xenopus", "yellow passion fruit", "yak", "zucchini", "zebra"

If we have the dictionary have the word that we are looking for:
Then return the index
If not, then “Word not found in dictionary”

Example Binary Search

```
public void searchWord(String word){
       int result = binarySearch(dict, word);
       if (result != -1) {
           UI.println("Word found at index: " + result);
       } else {
           UI.println("Word not found in dictionary.");
       }
   }
```

Example Binary Search

```
public static int binarySearch(String[] dict, String key) {
        int low = 0;
        int high = dict.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;

            int compare = dict[mid].compareTo(key.toLowerCase());

            if (compare == 0) {
                return mid; // found
            } else if (compare < 0) {
                low = mid + 1; // search right side
            } else {
                high = mid - 1; // search left side
            }
        }
        return -1; // not found
    }
```

$\log_2(n)$ or $\lg(n)$:

The **cost of binary search for input size n** is:
Time Complexity

$$O(log_2 n)$$

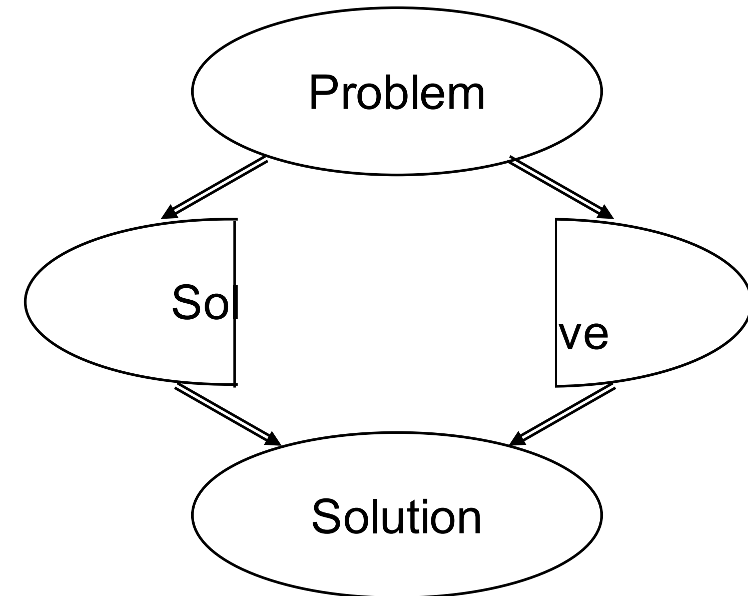
Why?
Binary search works by repeatedly dividing the search space into **half**:
•Start with n elements
•After 1 step → n/2
•After 2 steps → n/4
•After k steps → n/2^k
Stop when the search space becomes 1:

$$\frac{n}{2^k} = 1$$

Solve:

$$n = 2^k \Rightarrow k = log_2 n$$

Final answer
•**Best case:** $O(1)$) if the middle element is found immediately)
•**Average case:** $O(\log n)$
•**Worst case:** $O(\log n)$
So, the **cost of binary search is logarithmic:** $O(\log n)$.

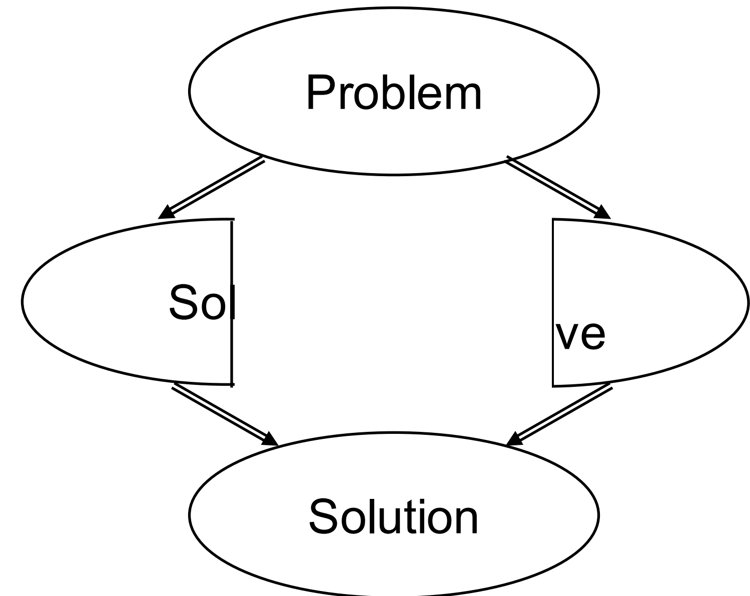


$\log_2(n)$ or $\lg(n)$:

The number of times you can divide a set of *n* things in half.

$\lg(1000) \approx 10$, $\lg(1{,}000{,}000) \approx 20$, $\lg(1{,}000{,}000{,}000) \approx 30$

Every time you double *n*, you add one *lg(n)*



- Arises all over the place in analysing algorithms
- "Divide and Conquer" algorithms
  - Good sorting algorithms
  - binary search (sort of)
- Height of binary trees:
  - Binary tree of height h has at most $2^h - 1$ nodes (h = number of levels)
  - Binary tree with n nodes has height at least $\log_2(n)$

Bisection Algorithm

The bisection method is a technique for finding solutions to equations with a single unknown variable.

Among various numerical methods, it stands out for its simplicity and effectiveness, particularly when dealing with **transcendental equations** (those that cannot be solved using algebraic methods alone). The method is also called the interval halving method, the binary search method or the dichotomy method.

This method is used to find the root of an equation in a given interval, that is, the value of 'x' for which $f(x) = 0$.

The bisection method is based on the Intermediate Value Theorem, which states that if $f(x)$ is a continuous function on the interval $[a, b]$ and $f(a)$ and $f(b)$ have opposite signs (i.e., $f(a)\cdot f(b)<0$), then there is at least one root of the equation $f(x)=0$ in the interval (a,b).

Bisection Algorithm

Assumptions:

1.f(x) is a continuous function on the interval [a, b].

2.f(a) ·f(b) < 0 (i.e., the function values at a and b have opposite signs).

Steps:

1.Find the middle point $c = \frac{a+b}{2}$.

2.If f(c) = 0, then c is the root of the equation.

3.If f(c) ≠ 0:

   3. If f(a)·f(c)<0, the root lies between a and c, so we recur with the interval [a, c].
   4. Else, if f(b)·f(c)<0, the root lies between b and ccc, so we recur with the interval [c, b].

4.If neither of these conditions hold, then the function does not satisfy the assumptions of the bisection method.

5.Since the root may be a floating-point number, we repeat the above steps until the difference between a and b is less than or equal to a very small value

Bisection Algorithm

Example: Finding the Root of a Polynomial

We use the bisection method to find the root of the polynomial: $f(x)=x^3-x-2$

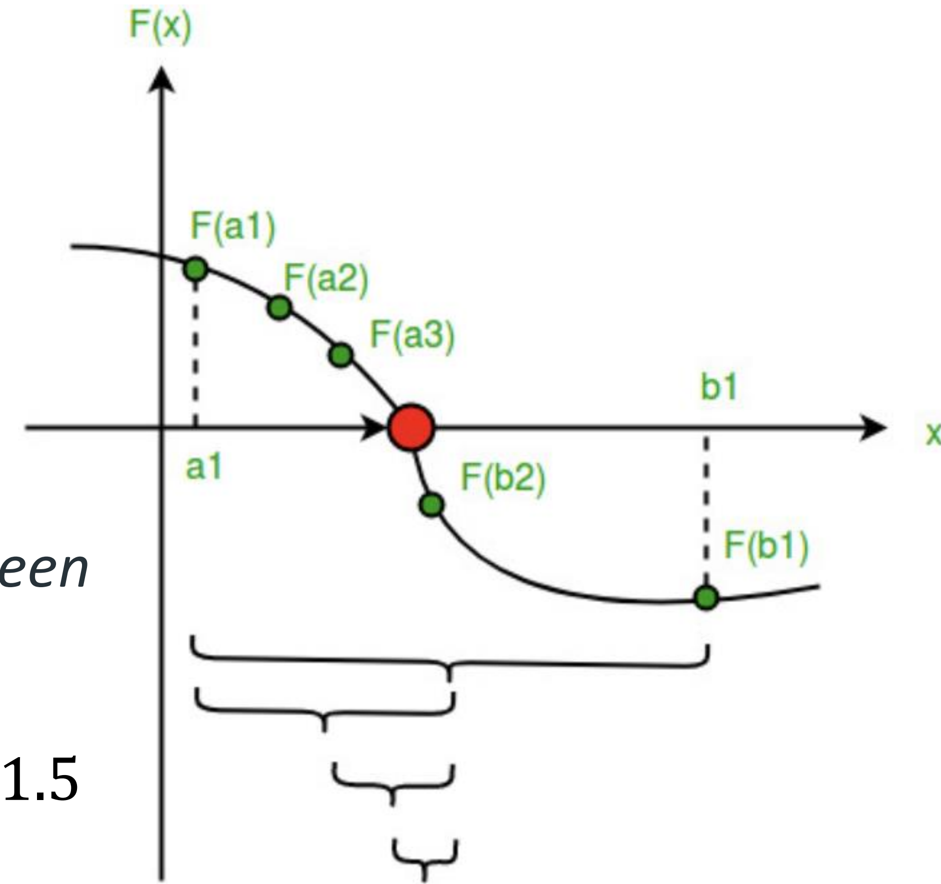
Step 1: Initial Interval Selection *Choose a= 1 and b= 2. Evaluate the function: f(1) = −2, f(2) = 4*

Since f(1) and f(2) have opposite signs, there is a root between 1 and 2.

Step 2: First Iteration*, Calculate the midpoint:* $c_1 = \frac{1+2}{2} = 1.5$
Evaluate the function at c_1 : f(1.5) = −0.125

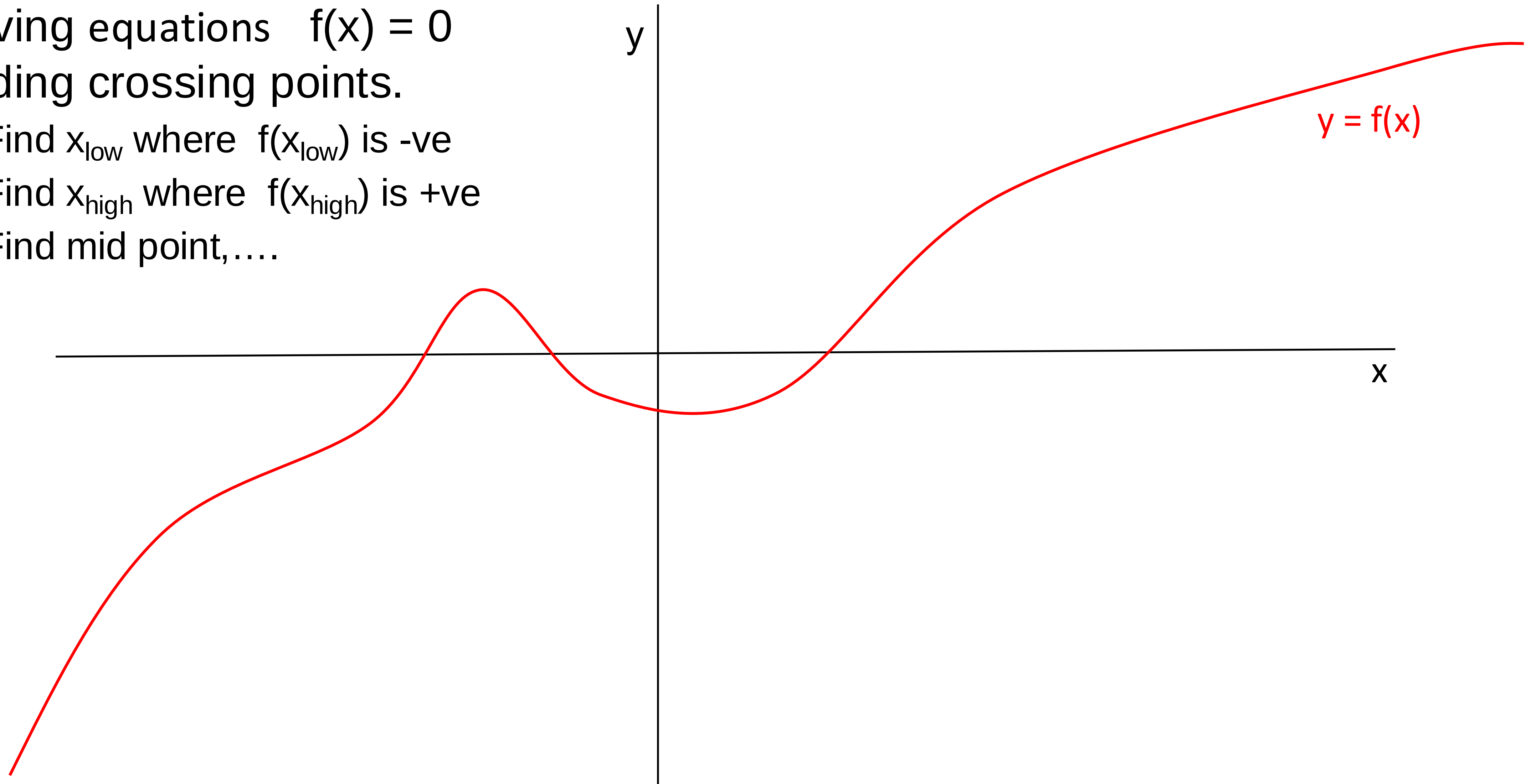
Since f(1.5) is negative, update the interval to [1.5, 2].

Step 3: Repeat*: Repeat the process until the interval becomes sufficiently small, converging on the root.*



Bisection Algorithm

- Solving equations $f(x) = 0$
  Finding crossing points.
  - Find x_{low} where $f(x_{low})$ is -ve
  - Find x_{high} where $f(x_{high})$ is +ve
  - Find mid point,….



Bisection Algorithm

Example

Determine the root of the given equation $x^2-3 = 0$ for $x\in[1,2]x$.

Solution:

Given: $x^2-3=0$

Let $f(x)=x^2-3$.

Now, find the value of f(x)at a=1 and b=2.

$f(1)=1^2-3=1-3=-2$ (which is <0),

$f(2)=2^2-3=4-3=1$ (which is >0)

The given function is continuous, and since $f(1)\cdot f(2) < 0$, the root lies in the interval [1, 2].

Let t be the midpoint of the interval: $t = \frac{1+2}{2} = \frac{3}{2} = 1.5$

Now, find the value of the function at t=1.5 :

f(1.5)=(1.5)2−3=2.25−3=−0.75 (which is <0)

Since $f(1.5) < 0$, we update $a = t = 1.5$.

Bisection Algorithm

Iterations for the Given Function:

At **iteration 7**, the final interval is [1.7266, 1.7344]. Hence, the approximated solution is **1.7344.**

Iteration	a	b	t	f(a)	f(b)	f(t)
1	1	2	1.5	-2	1	-0.75
2	1.5	2	1.75	-0.75	1	0.062
3	1.5	1.75	1.625	-0.75	0.0625	-0.359
4	1.625	1.75	1.6875	-0.359	0.0625	-0.1523
5	1.6875	1.75	1.7188	-0.1523	0.0625	-0.0457
6	1.7188	1.75	1.7344	-0.0457	0.0625	0.0081
7	1.7188	1.7344	1.7266	-0.0457	0.0081	-0.0189

Bisection

```
bisection( -100, 100, (double x)-> {return (3*x*x*x - 4*x*x + 321);} );
bisection( -100, 100, (x)-> (3*x*x*x - 4*x*x + 321) );

public double bisection(double low, Double high, Function<Double, Double> function){
    double fLow = function.apply(low);
    double fHigh = function.apply(high);
    if (Math.abs(fLow)<THETA)  { return fLow; }
    if (Math.abs(fHigh)<THETA) { return fHigh; }
    if (Math.signum(fLow) == Math.signum(fHigh)) { return Double.NaN; }  // same side of axis
    while (true) {
        double mid = (low+high)/2;
        double fMid = function.apply(mid);
        if (Math.abs(fMid)<THETA) {return mid;}
        else if (Math.signum(fLow) == Math.signum(fMid) ){  low = mid; fLow=fMid; }
        else {  high = mid;  fHigh=fMid; }
    }
}
```