Algorithms (Part 1) – Searching

If-then-else, loops, arrays, and functions work together.

What is "search"?
Given a search space, find the target, or see if the target exists

Outline

- Common data searching tasks on a data set (numeric or strings):

  #1: Look for the Minimum / Maximum?

  #2: Look for the i-th member? (e.g., i-th smallest/largest)

  #3: Query the existence of a specific value?

First of all, any idea to do so?
What is the algorithm or procedure?

Find the minimum

- Looking for the minimum item:
  - **Method 1**: if you know the input data range
  - **Strategy**: start "**min**" with a value bigger than any item

```c
int main( void )
{
    double array[ 5 ] = { 1.1 , 191.9 , 34 , 0 , 999.9 } ;
    double min        = 1001 ; // bigger than any possible value
    int    len        = 5    ;

    for ( int i = 0 ; i < len ; i++ )
        if ( min > array[ i ] )
            min = array[ i ] ;

    printf( "Minimum = %f\n" , min );
    return 0 ;
}
```

Key idea? Keep track of the smallest value that the procedure has seen so far

Find the minimum

- Looking for the minimum item:
  - **Method 2**: if you don't know the input data range
  - **Strategy**: start "**`min`**" with the first array element

```c
int main( void )
{
    double array[ 5 ] = { 1.1 , 191.9 , 34 , 0 , 999.9 } ;
    double min        = array[ 0 ] ;   // the first item
    int    len        = 5 ;

    for ( int i = 1 ; i < len ; i++ )   // skip the first item
        if ( min > array[ i ] )
            min = array[ i ] ;

    printf( "Minimum = %f\n" , min );
    return 0 ;
}
```

Find the minimum

- Looking for the minimum item:
  - **Method 3**: not sure if there is at least one data value?
  - **Strategy**: start "**min**" with the largest double value

```c
int main( void )
{
    double array[ 5 ] = { 1.1 , 191.9 , 34 , 0 , 999.9 } ;
    double min        = DBL_MAX ;   // include <float.h>
    int    len        = 5 ;

    for ( int i = 0 ; i < len ; i++ )   // skip the first item
        if ( min > array[ i ] )
            min = array[ i ] ;

    printf( "Minimum = %f\n" , min );
    return 0 ;
}
```

Find the minimum

- What if there are only **two items**?
  - Simple:

```
double a = 10 , b = 11 , min ;

if ( a < b )
    min = a ;
else
    min = b ;

printf( "Minimum = %f\n" , min );
```

  - Simpler:

```
double a = 10 , b = 11 , min ;

min = ( a < b ) ? a : b ;

printf( "Minimum = %f\n" , min );
```

?: - ternary conditional operator

- Ternary operator: an **n**-ary operator, where **n = 3**
  - Syntax:

`Input 1 (must be int)`	`?`	`Input 2 (any type)`	`:`	`Input 3 (any type)`

  - If **Input 1 is non-zero**, the operator **returns Input 2**. Else, the operator **returns Input 3**.

?: - ternary conditional operator

- They are the same!

```
double min = ( input1 < input2 ) ? input1 : input2 ;
```

```
double minimum( double a , double b )
{
    if ( a < b )
        return a ;
    else
        return b ;
}

double min = minimum( input1 , input2 );
```

How about this?

```
double min =       ( input1 < input2 )   * input1
           + ( 1 - ( input1 < input2 ) ) * input2 ;
```

But… which one is faster?

Find the maximum

- **<u>Looking for the maximum</u>** is similar to **<u>looking for the minimum</u>**.

How to change the code to look for max instead of min?

```
int main( void )
{
    double array[ 5 ] = { 1.1 , 191.9 , 34 , 0 , 999.9 } ;
    double min        = array[ 0 ] ;    // the first item
    int    len        = 5     ;

    for ( int i = 1 ; i < len ; i++ )    // skip the first item
        if ( min > array[ i ] )
            min = array[ i ] ;

    printf( "Minimum = %f\n" , min );
    return 0 ;
}
```

Find the maximum

- **<u>Looking for the maximum</u>** is similar to **<u>looking for the minimum</u>**.

```c
int main( void )
{
    double array[ 5 ] = { 1.1 , 191.9 , 34 , 0 , 999.9 } ;
    double max        = array[ 0 ] ;   // the first item
    int    len        = 5     ;

    for ( int i = 1 ; i < len ; i++ )   // skip the first item
        if ( max < array[ i ] )
            max = array[ i ] ;

    printf( "Maximum = %f\n" , max );
    return 0 ;
}
```

Outline

- Common data searching tasks on a data set (numeric or strings):

  #1: Look for the Minimum / Maximum?

  #2: Look for the i-th member? (e.g., i-th smallest/largest)

  #3: Query the existence of a specific value?

Sorting – What is it?

- "**<u>In general</u>**", we can sort the data…

Goal: **<u>order (re-arrange) the data values in an array</u>** in

<u>Ascending order</u>:

- For all i > 0, `array[i-1] <= array[i]`

OR

<u>Descending order</u>:

- For all i > 0, `array[i-1] >= array[i]`

Selection sort algorithm [Example]

```
int A[ 6 ] = { 4 , 5 , 7 , 2 , 9 , 1 };
```

Iteration #	Sub-array to be processed: A[i] … A[N-1]	Locate the smallest item in sub-array	Swap the smallest item with A[i]
i = 0	4 5 7 2 9 1	4 5 7 2 9 1	1 5 7 2 9 4
i = 1	1 5 7 2 9 4	1 5 7 2 9 4	1 2 7 5 9 4
i = 2	1 2 7 5 9 4	1 2 7 5 9 4	1 2 4 5 9 7
i = 3	1 2 4 5 9 7	1 2 4 5 9 7	1 2 4 5 9 7 NO SWAP
i = 4	1 2 4 5 9 7	1 2 4 5 9 7	1 2 4 5 7 9

Selection sort algorithm [Code]

```
// Sort an array of N elements, so that a[i] <= a[i+1]
void selection_sort( int a[] , int N )
{
    int minPos , tmp ;

    for ( int i = 0 ; i <        ; i++ )
    {
        // Find the position of the next minimum number
        minPos = i ;   // let's first take a[i] as the min. value
        for ( int j =        ; j        ; j++ )
            if ( a[ minPos ] > a[ j ] )
                minPos = j ;

        // Swap a[i] with a[minPos] if necessary
        if ( minPos != i )
        {
                                              ;
                                              ;
                                              ;
        }
}   }
```

[Complete Code]

```
// Sort an array of N elements, so that a[i] <= a[i+1]
void selection_sort( int a[] , int N )
{
    int minPos , tmp ;

    for ( int i = 0 ; i < N - 1 ; i++ )
    {
        // Find the position of the next minimum number
        minPos = i ;   // let's first take a[i] as the min. value
        for ( int j = i + 1 ; j < N ; j++ )
            if ( a[ minPos ] > a[ j ] )
                minPos = j ;

        // Swap a[i] with a[minPos] if necessary
        if ( minPos != i )
        {
            tmp          = a[ i ]      ;
            a[ i ]       = a[ minPos ] ;
            a[ minPos ] = tmp          ;
        }
}   }
```

Sidetrack: pass an array to functions

- **Important**: pass an array into a function:

```
void selection_sort( int a[] , int N )
```

- But, array is a **special** storage:
  - After processed by **`selection_sort()`**, the caller will find the array elements changed (re-arranged)!
  - **The content of the array will be updated if the function (callee) updates the array**.

Sidetrack: pass an array to functions

```
void print_array( int a[] , int len )
{
    for ( int i = 0 ; i < len ; i++ )
        printf( "%d " , a[ i ] );
    printf( "\n" );
    len = 0 ;   // len is not an array; won't affect the caller
}               // it is in fact a **local variable**

int main( void )
{
    int array[ 6 ] = { 4 , 5 , 7 , 2 , 9 , 1 };
    int size       =   6 ;

    print_array    ( array , size ) ;
    selection_sort ( array , size ) ;  // array is updated!
    print_array    ( array , size ) ;

    return 0 ;
}
```

Sidetrack Question: Discussion

- If we want to find the **k-th largest number** in a list of integers…… How to do?
- Do we really need to sort the data first?
- Which way is faster? Searching a sorted array? Or searching an unsorted array?
  - It really depends, e.g.,
    - What is the value of k? If k is just a small number, say 2 or 3?
    - Do you need to search many times? **Hint**: use a small heap!

[As a **self exercise** – find the **median** without sorting; **hint**: modify the well-known "quick sort" algorithm]

Outline

- Common data searching tasks on a data set (numeric or strings):

  #1: Look for the Minimum / Maximum?

  #2: Look for the i-th member? (e.g., i-th smallest/largest)

  #3: Query the existence of a specific value?

Search an array

- What is searching?
  - **Goal**. Look up a value in the input array
  - **Result**. Two kinds of implementations:

    1. Decision problem:
       - **Return 1 (true)** when the target value is found
       - **Return 0 (false)** otherwise

    2. Location problem:
       - **Return the index of the target value in the array** if there is more than one value that matches the target value, return the index of the first encountered value
       - **Return -1** if the target value is not found

Search an array – Sequential

- Sequential search: **decision**

```
int seq_search( int array[] , int len , int target )
{
    for ( int i = 0 ; i < len ; i++ )
        if ( array[ i ] == target )
            return 1 ;    // target found

    return 0 ;             // target not found
}
```

Search an array – Sequential

- Sequential search: **location**

```
int seq_search( int array[] , int len , int target )
{
    for ( int i = 0 ; i < len ; i++ )
        if ( array[ i ] == target )
            return i ;    // target index array

    return -1 ;           // -1 is never an array index
}
```

Search an array – Sequential

- Computer scientists are interested to know how a program performs: **<u>How fast is a program?</u>**

- Let's analyze the performance of sequential search:
  - Best-case, worst-case, & average-case scenarios

- Performance metrics:
  <u>the number of comparisons performed</u> (say **C**)
  before we get the answer

Sequential search – Analysis

- What is the best case?
  - Input:

Array	1	2	3	4	5

  - Search target: 1
  - C = 1

```
for ( i = 0 ; i < len ; i++ )
   if ( array[ i ] == target )
      return i ;
return -1 ;
```

- What is the worst case?
  - Input:

Array	1	2	3	4	5

  - Search target: 5
  - C = 5

Sequential search – Analysis

- What is the average case?
  - Let's randomly choose a number in [1,5] as the target.
  - Each target is assumed to have an equal probability to be chosen, i.e., 1/5
  - Input:

Array	1	2	3	4	5

  - If target is 3, we say **C = 3 (each num. has same prob.)**

$$\mathbf{C} = 1 \times \frac{1}{5} + 2 \times \frac{1}{5} + 3 \times \frac{1}{5} + 4 \times \frac{1}{5} + 5 \times \frac{1}{5}$$

Sequential search – Analysis

- Let C_n be the number of comparisons over an array of size **n**
  - $C_n = k$ means the target element is in the **k**-th position in the array, where 1 <= k <= n.
  - Let's also assume that the target is always in the array.

A.k.a. **Expected Value** in statistics

$$
\begin{aligned}
E[C_n] &= \sum_{k=1}^{n} P(C_n = k) \times k \\
&= \frac{1}{n}\sum_{k=1}^{n} k \\
&= \frac{1}{n} \times \frac{n \times (n+1)}{2} \\
&= \frac{n+1}{2}
\end{aligned}
$$

probability

```
Average Case: C = (n+1) / 2
```

Sequential search – Analysis

- In computer science, we are typically interested in the **worst-case scenario**.

Best Case	`1`
Average Case	`(n+1) / 2`
Worst Case	`n`

- We call "the running time of the sequential search algorithm is of **order n**", or a **linear-time algorithm**.

Is it possible to search faster?

Search an array – Binary search

- If an array is **sorted**, we can use a faster search implementation: **binary search**.

(0) Define **left = 0** and **right = array-length – 1 (search range)**

(1) Look at the middle element in range [left, right]

(2) If it is the **target** (or **left > right**) then

    stop // found (or not found)

(3) Else if **target** is smaller than the middle element, then

    right = middle - 1

(4) Else if **target** is larger than the middle element, then

    left = middle + 1

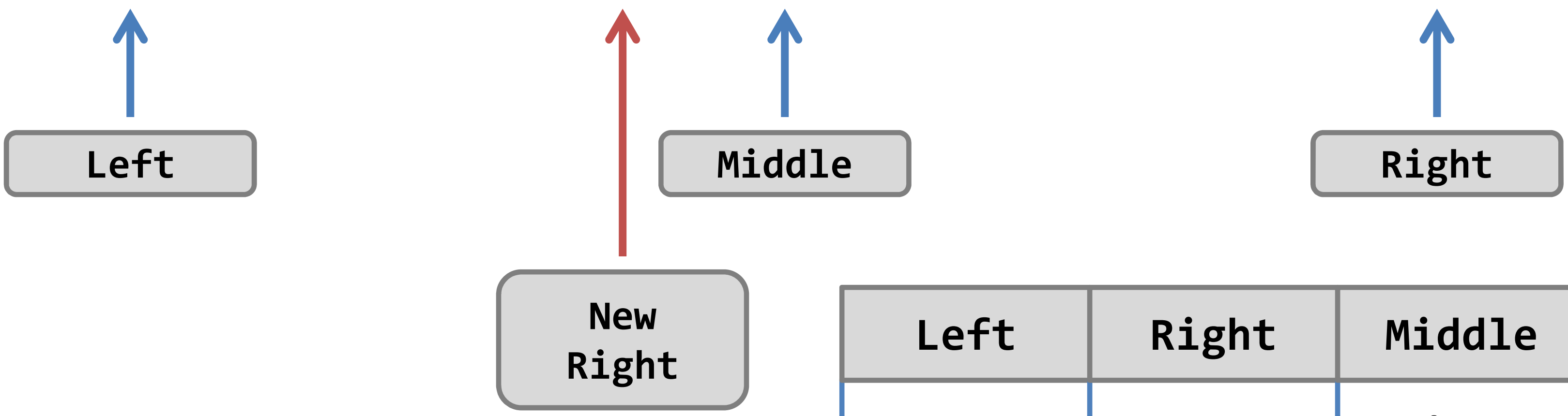
(5) Go back to (1)

Binary search – illustration #1

- Target = 5:

Round 1

Index	0	1	2	3	4	5	6	7	8
Value	5	6	10	21	77	200	689	1999	2000



Left	Right	Middle
0	8	8/2 = 4

Binary search – illustration #1

- Target = 5:

Round 2

New Right

Index	0	1	2	3	4	5	6	7	8
Value	5	6	10	21	77	200	689	1999	2000

Left

Middle

Right

Left	Right	Middle
0	3	3/2 = 1

Binary search – illustration #1

- Target = 5: Found!

Round 3

Index	0	1	2	3	4	5	6	7	8
Value	5	6	10	21	77	200	689	1999	2000

Left
& Right
& Middle

Left	Right	Middle
0	0	0/2 = 0

Binary search – illustration #2

- Target = 2000:

Round 1

New Left

Index	0	1	2	3	4	5	6	7	8
Value	5	6	10	21	77	200	689	1999	2000

Left

Middle

Right

Left	Right	Middle
0	8	8/2 = 4

Binary search – illustration #2

- Target = 2000:

Round 2

New Left

Index	0	1	2	3	4	5	6	7	8
Value	5	6	10	21	77	200	689	1999	2000

Left

Right

Middle

Left	Right	Middle
5	8	13/2 = 6

Binary search – illustration #2

- Target = 2000:

Round 3

New Left

Index	0	1	2	3	4	5	6	7	8
Value	5	6	10	21	77	200	689	1999	2000

Right

Left
& Middle

Left	Right	Middle
7	8	15/2 = 7

Binary search – illustration #2

- Target = 2000: Found!

Round 4

Index	0	1	2	3	4	5	6	7	8
Value	5	6	10	21	77	200	689	1999	2000

Left
& Right
& Middle

Left	Right	Middle
8	8	16/2 = 8

Binary search – Implementation

```
int bin_search( int array[] , int target , int left , int right )
{
    while ( left <= right )
    {
        int mid = ( left + right ) / 2 ;

        if ( array[ mid ] == target )
            return mid ;                    // found it!!!
        else
        if ( array[ mid ] >  target )
            right = mid - 1 ;
        else
            left  = mid + 1 ;
    }
    return -1 ;
}
```

Binary search – Analysis

- Back to previous illustration:

Round 1	5	6	10	21	77	200	689	1999	2000
Round 2	5	6	10	21	77	200	689	1999	2000
Round 3	5	6	10	21	77	200	689	1999	2000
Round 4	5	6	10	21	77	200	689	1999	2000

- After each round of iteration, the search range is halved, more precisely: $\left\lfloor \frac{length}{2} \right\rfloor$

Binary search – Analysis

- Let
  - **n** be the length of the array and let **n** be a power of 2
  - **k** be the number of iterations for range to reduce to 1

$$
\begin{aligned}
\frac{n}{2^k} &= 1 \\
2^k &= n \\
k &= log_2 n
\end{aligned}
$$

Question:

What if n is not a power of 2?
Then, what is k?

Binary search – Analysis

- Any assumption in our binary search algorithm?
- Think carefully…

What if the data values are not unique?
- Even the data is sorted, several data values may be the same. So?

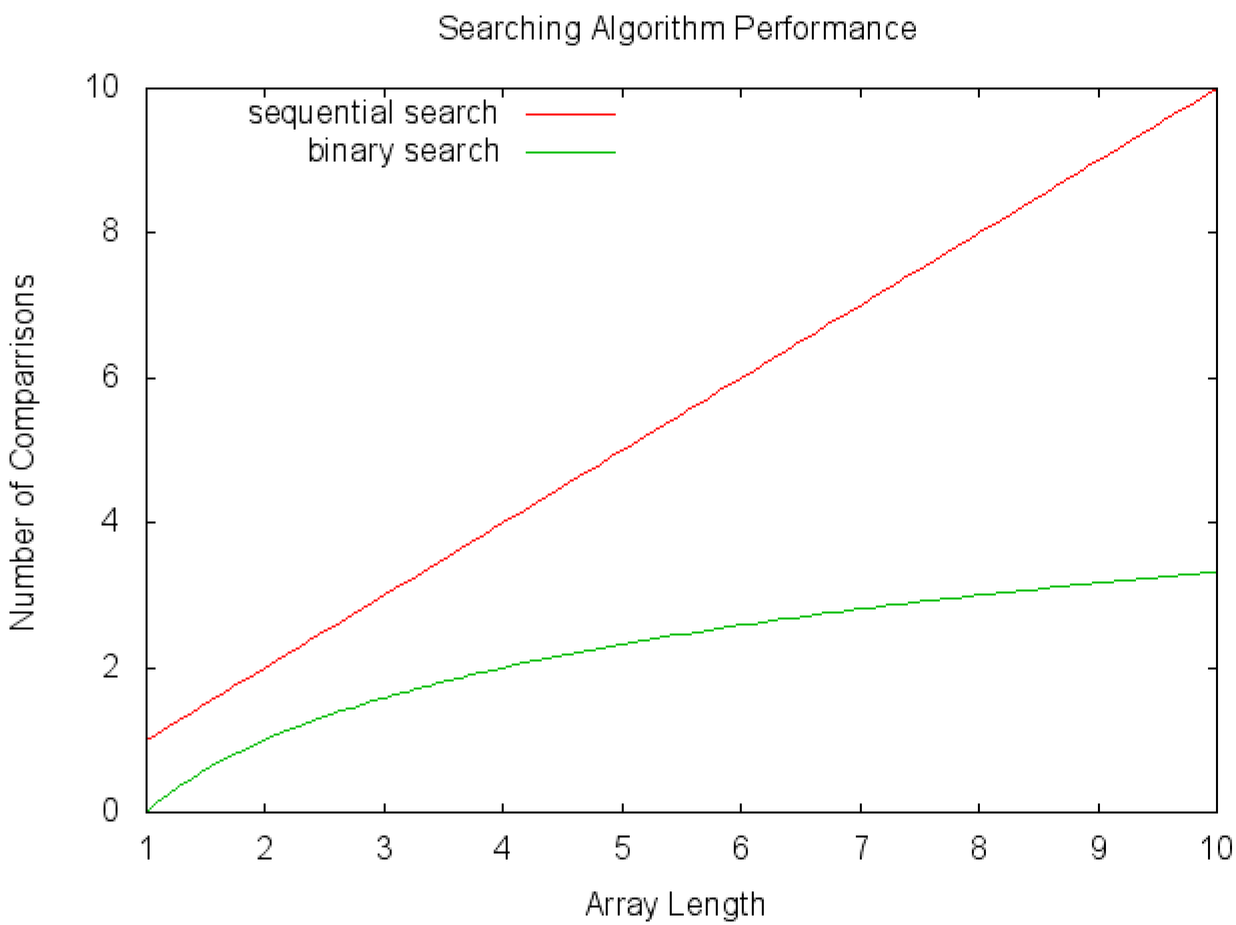
How if we need to answer a more complex question…
- Is value t inside the data?
- If inside, how many of them in the data?
  e.g., how many students get 75 marks?

Searching algorithm – Summary

- Sequential search has a running time of **order n**

- Binary search is a much faster algorithm of running time of **order $\log_2 n$**
  – However, you have to sort the entire array before you can use the binary search algorithm.

Searching algorithm – Summary



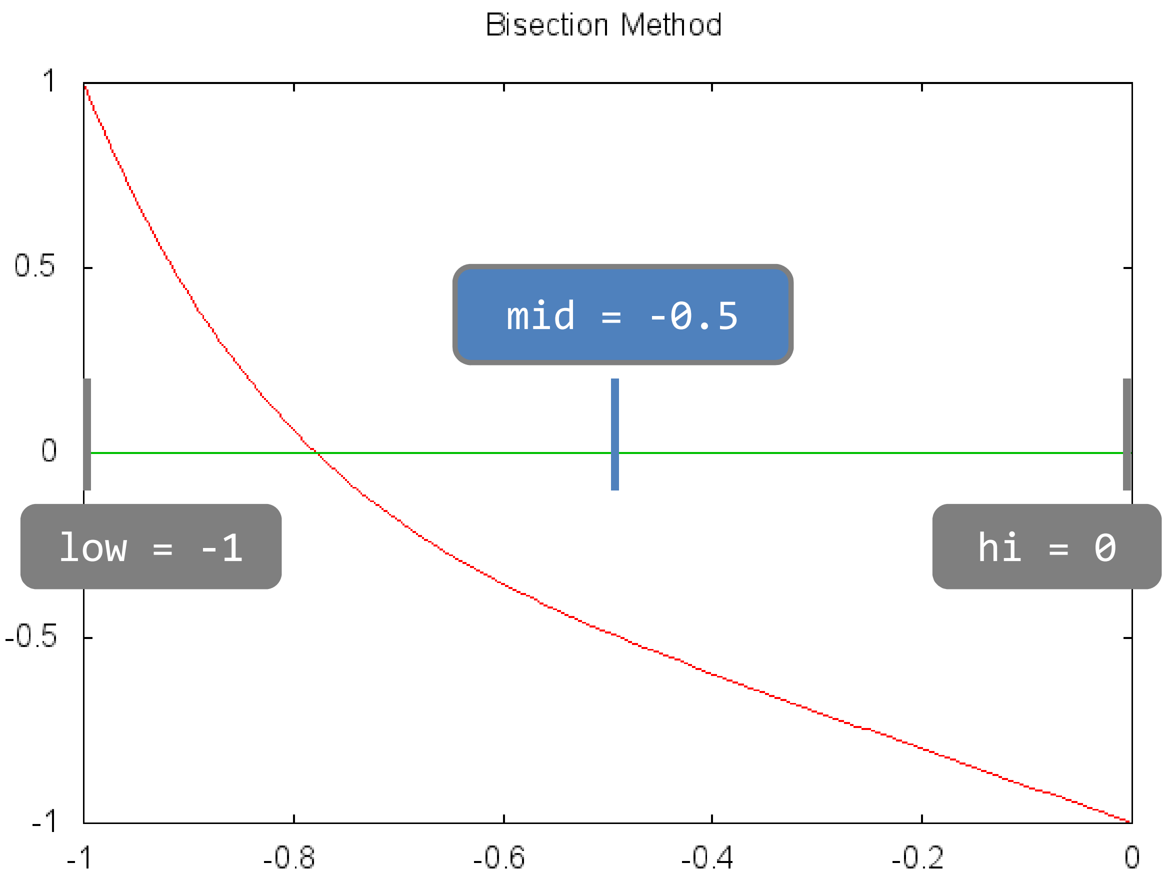
As the array's size grows, so does this gap

Sidetrack – Root Finding

- A well-defined field of study in solving non-linear Eqn.
  - Related course in Math department: http://www.math.cuhk.edu.hk/course/1516/math3230a
- Let's look at **bisection method**, which is a simple method very similar to binary search!
- Task:

  Solve $\mathbf{f(x) = x^6 - x - 1 = 0}$ for $\mathbf{x} \in \mathbf{[-1,0]}$
  - Note: `f(-1) = +1, f(0) = -1, and f(x) is continuous, so f(x) must go through x-axis`

Bisection method



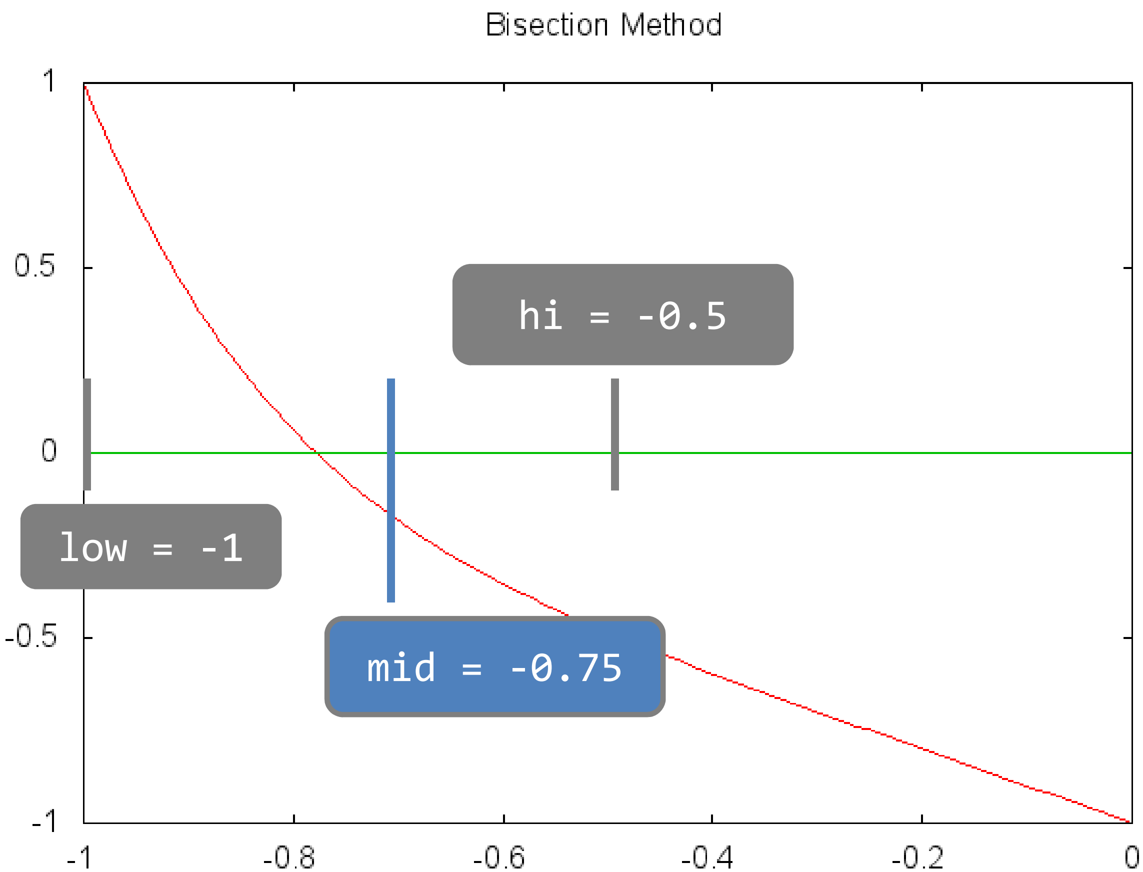
<u>Step 1:</u>

Set low = 0
Set high = 1

`F(low)` and **`F(high)`** must have opposite signs
(To you: why?)

Calculate the midpoint:
"**`mid = (low+high)/2`**"

Bisection method



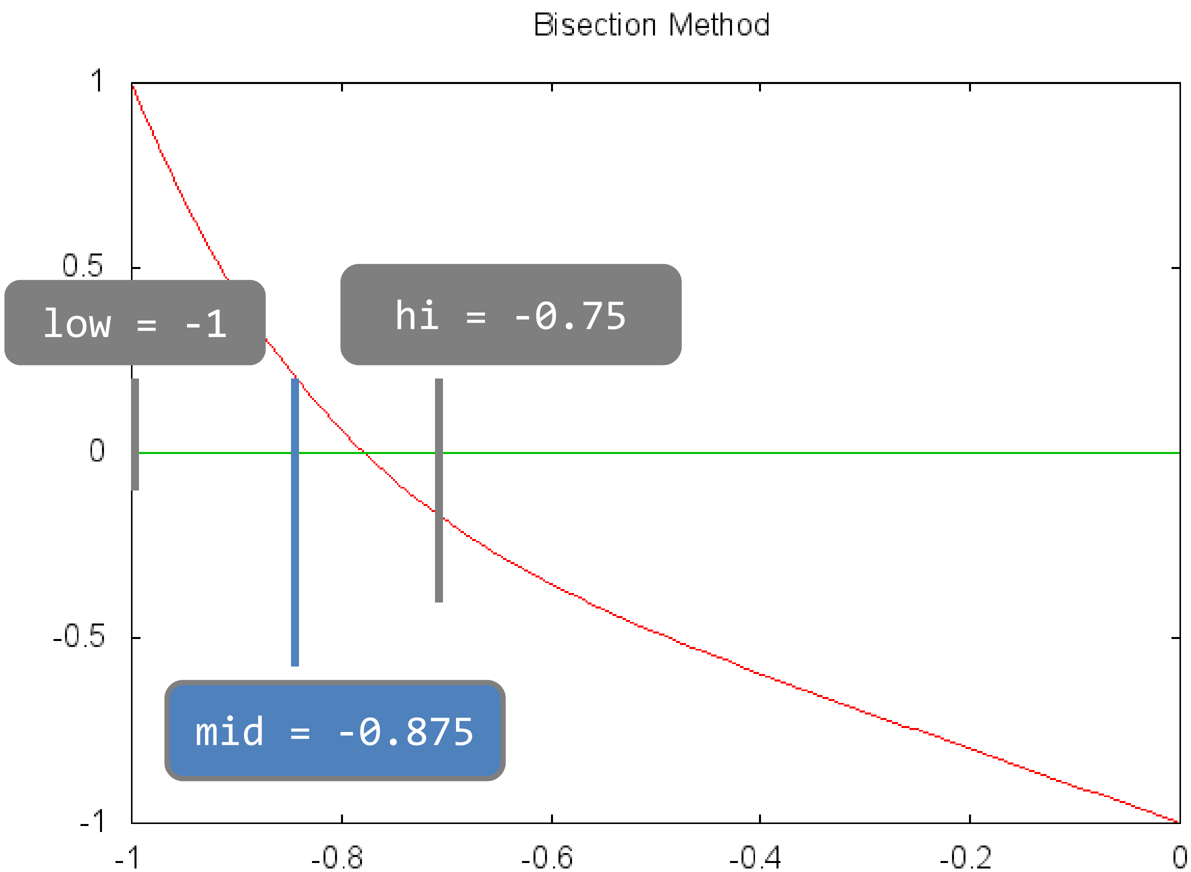
Step 2:

Evaluate F(mid)

Since
`sign of F(mid) ==`
`sign of F(hi) ,`
set **high = mid**

Calculate the midpoint:
"**mid = (low+high)/2**"

Bisection method



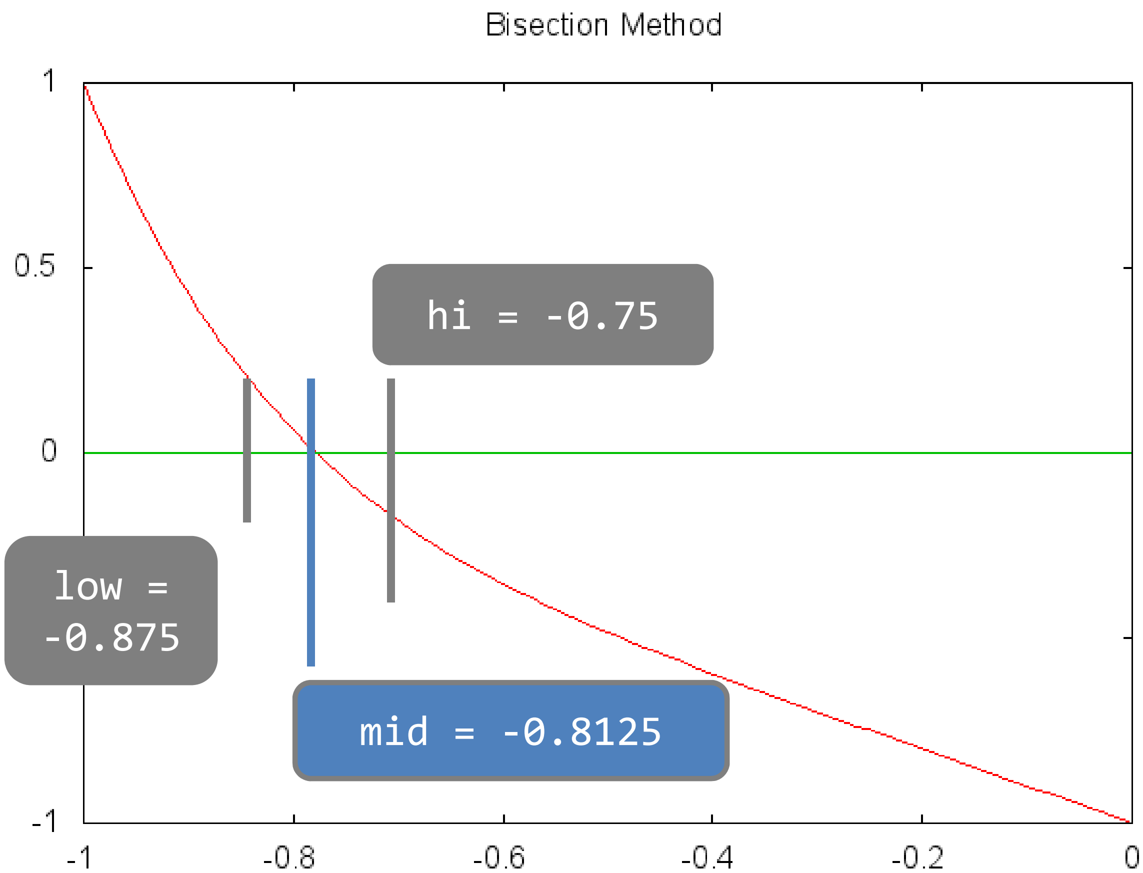
Step 3:

Evaluate `F(mid)`

Since
`sign of F(mid) ==`
`sign of F(hi) ,`
set **`high = mid`**

Calculate the midpoint:
"**`mid = (low+high)/2`**"

Bisection method



Step 4:

Evaluate `F(mid)`

Since
`sign of F(mid) ==`
`sign of F(low) ,`
set **`low = mid`**

Calculate the midpoint:
"**`mid = (low+high)/2`**"

Bisection method

Step	low	f(low)	high	f(high)	mid	f(mid)
1	-1	+	0	-	-0.5	-
2	-1	+	-0.5	-	0.75	-
3	-1	+	-0.75	-	-0.875	+
4	-0.875	+	-0.75	-	-0.8125	+
5	-0.8125	+	-0.75	-	-0.78125	+
6	-0.78125	+	-0.75	-	-0.765625	-
7	-0.78125	+	-0.765625	-	...	...

Question: When should the process stop?

Answer: Depending on the precision you want!

Bisection method – Bound

- Let *a* and *b* be the initial low and initial high points, respectively

- Let a_n and b_n be the low and high points at the *n*-th iteration, respectively

$$\begin{aligned} b_{n+1} - a_{n+1} &= \frac{1}{2}(b_n - a_n) \\ \Rightarrow \; b_n - a_n &= \frac{1}{2^{n-1}}(b - a) \end{aligned}$$

Bisection method – Bound

- Let α be the true root
- Let c_n be the mid-point obtained in n-th iteration

$$|\alpha - c_n| \leq \frac{1}{2^n}(b-a)$$

$\therefore$ As n $\rightarrow \infty$, the difference tends to 0.

Bisection method – Bound

- Say, the bound that we want to achieve is ε.

$$
\begin{aligned}
&|\alpha - c_n| \leq \epsilon \\
\Rightarrow\ & \frac{1}{2^n}(b-a) \leq \epsilon \\
\Rightarrow\ & 2^n \geq \frac{b-a}{\epsilon}
\end{aligned}
$$

- Taking log_2 on both sides,

$$n \geq log_2\left(\frac{b-a}{\epsilon}\right)$$

Bisection method – Bound

- Given a = -1 and b = 0,
- How many iterations are needed if we want to obtain a result up to **4 decimal places**?

Round to 4 decimal places: $\epsilon = 0.0001$.

$$
\begin{aligned}
n &\geq log_2\left(\frac{b-a}{\epsilon}\right) = log_2\left(\frac{1}{0.0001}\right) \\
\Rightarrow \quad n &\geq 13.2877
\end{aligned}
$$

$\therefore$ At least 14 iterations