Fundamental of Computer Science
CS1FC16: Lecture 03

# Searching

Dr Varun Ojha

Department of Computer Science

University of Reading

# Learning Objectives

On completion of three parts of this lecture, you will be able to

- Understand how to write a search algorithm
- Evaluate complexity of search algorithms
- Write recursive binary algorithm for faster search

# Content of this lecture

- Part – I: Linear search
  - Search algorithms basics
  - Linear search
- Part – II: Binary search
  - Binary search
  - Recursive binary search
- Part –III
  - Exercises

Fundamental of Computer Science
CS1FC16: Lecture 03, Part – I

# Linear Search

Dr Varun Ojha
Department of Computer Science

University of Reading

# Search

```
search(datum, datastructure) -> (index, found)
```

- A search algorithm takes a datum (key element) and a datastructure (input data) as arguments
- If the datum is in the datastructure then it returns an index at which the datum can be found in the datastructure
- If the datum is not in the datastructure then it returns a special value to indicate this fact

# Selection

- A selection algorithm searches for a datum with a desired property
- Find the median of a list of items
- Find the element equal to or just greater than the average of a list of numbers
- Find an existing customer who is likely to spend £1000 in the coming year

# Matching

- A matching algorithm searches for a pattern in a datastructure
- Find the position of “sub” in “this is a substring”
- Find consecutive [1,0,1] in a list [1,2,0,1,0,1,1,1,0,2,1,0,1].
- Patterns might you like to search for

# Exhaustive Search

- Search every element of a datastructure to see if it is the target element
- For example, search a sequence of elements
- How would you arrange a parallel search algorithm?
- The most general algorithm in Computer Science is generate and test. It generates a space of candidate solutions, tests each element in the space to see if it is a solution and then reports its findings

# Indexes

- If data are sorted in some way, then it may be possible to develop more efficient search algorithms
- Search an alphabetical index
- What is the collation sequence for digits, upper case letters, lower case letters, punctuation?
- Databases typically spend a great deal of time constructing many indexes of search keys so that searches will be efficient

# Linear Search

```
Input: datum, datastructure

Output: (index, found)

LinearSearch(datum, datastructure)
    index := undef
    found := false
    for i from 1 to length(datastructure) do
        d = datastructure(i)
        if d = datum then
              index := i
              found := true
              return
```

# Linear Search

```
Input: datum, datastructure

Output: (index, found)

LinearSearch(datum, datastructure)
    index := undef
    found := false
    for i from 1 to
    length(datastructure) do
        d = datastructure(i)
        if d = datum then
                index := i
                found := true
                return
```

**Questions:**

- What is the worst-case time of this algorithm?
- What is the average time of this algorithm?
- What is the best-case time of this algorithm?

# Linear Search: Average Time

```
Input: datum, datastructure

Output: (index, found)

LinearSearch(datum, datastructure)
    index := undef
    found := false
    for i from 1 to
    length(datastructure) do
        d = datastructure(i)
        if d = datum then
            index := i
            found := true
            return
```

- Suppose that a sequence is indexed from $i = 1$
- Suppose that processing the element in position takes $2i$ operations
- Let $T(n)$ be a function that returns the average time, $t$, that it takes to process $n$ index positions such that the integer $n$ is positive. Then:

$$T(n) = \frac{1}{n}\sum_{i=1}^{n} 2i, \text{ for all } n > 0, n \in \mathbb{Z}$$

# Linear Search: Average Time

```
Input: datum, datastructure

Output: (index, found)

LinearSearch(datum, datastructure)
    index := undef
    found := false
    for i from 1 to
    length(datastructure) do
        d = datastructure(i)
        if d = datum then
              index := i
              found := true
              return
```

$$T(n) = \frac{1}{n}\sum_{i=1}^{n} 2i$$

$$= \frac{1}{n}\left(2 \times \frac{n(n+1)}{2}\right)$$

$$= \frac{1}{n}\big(n(n+1)\big)$$

$$= n + 1$$

$$T(n) = O(n)$$

Fundamental of Computer Science
CS1FC16: Lecture 03, Part – II

# Binary Search

Dr Varun Ojha
Department of Computer Science
University of Reading

# Binary Search

- Input an ordered sequence of real numbers – an `array`
- Cut the sequence in half
- Decide which half the `target` datum lies in
- Recurse until the target has been identified or excluded

# Binary Search

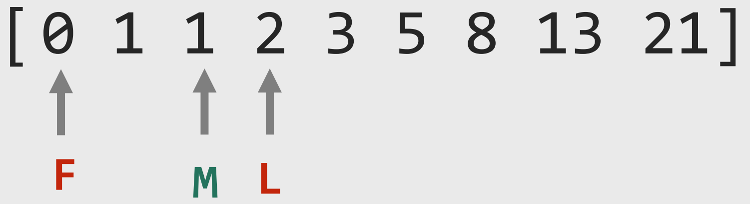
```
Input: target, array
Output: (index, found)
BinarySearch(target, array)
    first := 0
    last := length(array) - 1
    while first <= last do
        middle = (first + last)/2
        if array(middle) = target then
                return(middle, true)
        elseif array(middle) > target then /* Search left. */
                last := middle - 1
        else /* Search right. */
                first := middle + 1
        endif
    endwhile
    return(index, false) /* target not found */
```

# Binary Search: Numeric Example

```
Input: target, array
Output: (index, found)
BinarySearch(target, array)
    first := 0 /* F */
    last := length(array) - 1 /* L */
    while first <= last do
        middle = (first + last)/2 /* M */
        if array(middle) = target then
                return(middle, true)
        elseif array(middle) > target then
                last := middle - 1
        else
                first := middle + 1
        endif
    endwhile
    return(index, false)
```

Find in, target = 1 in

array [0 1 1 2 3 5 8 13 21]


index of ‘1’= 2, found true

# Binary Search: Complexity

```
Input: target, array
Output: (index, found)
BinarySearch(target, array)
    first := 0 /* F */
    last := length(array) - 1 /* L */
    while first <= last do
        middle = (first + last)/2 /* M */
        if array(middle) = target then
                return(middle, true)
        elseif array(middle) > target then
                last := middle - 1
        else
                first := middle + 1
        endif
    endwhile
    return(index, false)
```

**Worst case time complexity.**

- Let $n = 2^k$ be the number of elements with integral $k > 0$
- Initially there are $n = 2^k$ elements to be searched
- After step 1 there are $\frac{n}{2} = 2^{k-1}$ elements
- After step 2 there are $\frac{n}{2^2} = 2^{k-2}$ elements
- After step $k$ there is $\frac{n}{2^k} = 2^{k-k} = 2^0 = 1$ element. Hence, algorithm terminates

# Binary Search: Complexity

```
Input: target, array
Output: (index, found)
BinarySearch(target, array)
    first := 0 /* F */
    last := length(array) - 1 /* L */
    while first <= last do
        middle = (first + last)/2 /* M */
        if array(middle) = target then
                return(middle, true)
        elseif array(middle) > target then
                last := middle - 1
        else
                first := middle + 1
        endif
    endwhile
    return(index, false)
```

**Worst case time complexity.**

- Algorithm terminates after $k$ step (partitions)
- So, the time order (complexity), measured in the number of steps, $k$, varies with the number of elements, $n$, as:
    - $2^k = n$
    - $\log 2^k = \log n$
    - $k = \log n$
- Thus, binary search has time order $O(\log n)$

# Recursive Binary Search

```
Input: target, array

Output: (index, found)

BinarySearchRecursive(T, A(F,…,L)) /* T indicate target, A(F,…, L) is a sorted array of numbers */
    if F <= L then /* F indicate first, L indicate last */
        M = (F + L)/2 /* M indicate middle */
        if A(M)= T then
            return(M, true)
        if A(M) > T then
            BinarySearchRecursive(T, A(F,…,M-1)) /* A(F - M-1) -> search left */
        if A(M) < T then
            BinarySearchRecursive(T, A(M+1,…,L)) /* A(M+1 - L) -> search right */
    else
        return(index, false)
    endif
```

# Recursive Binary Search: Complexity

```
                Input: target, array
                Output: (index, found)
T(n)    ←       BinarySearchR(T, A(F,…,L))
                    if F <= L then
T(1)    ←               M = (F + L)/2
                        if A(M)= T then
                            return(M, true)
                        if A(M) > T then
T(n/2)  ←                   BinarySearchR(T, A(F,…,M-1))
          Or            if A(M) < T then
T(n/2)  ←                   BinarySearchR(T, A(M+1,…,L))
                    else
T(0)    ←               return(index, false)
                    endif
```

- $T(n)$ denotes the time complexity of binary search for an input of size an input of size $n$
- $T(n) = 0$, i.e., target does not present
- $T(n) = 1$, i.e., smallest sequence user can supply
- $T(n) = 1 + T(n/2)$
- Let $n = 2^k$,
- Solve the following recurrence relation:

$$T(n) = T(2^k)$$

# Recursive Binary Search: Complexity

- $T(n) = T(2^k)$
- $= 1 + T(2^{k-1})$ /* after step 1 */
- $= 1 + 1 + T(2^{k-1-1})$
- $= 2 + T(2^{k-2})$ /* after step 2 */
- $= 2 + 1 + T(2^{k-2-1})$
- $= 3 + T(2^{k-3})$ /* after step 3 */
- $\ldots$
- $= k + T(2^{k-k})$ /* after step $k$ */
- $= k + T(2^0)$
- $= k + 1$

- $T(n) = k + 1$
- **We know**
- $2^k = n$
- Therefore, $k = \log_2 n$
- $\boldsymbol{T(n) = \log_2 n + 1}$
- Thus, $\boldsymbol{T(n) = O(\log_2 n)}$

Fundamental of Computer Science
CS1FC16: Lecture 03, Part – III

# Exercises

Dr Varun Ojha
Department of Computer Science

University of Reading

# Exercises

Write a program in C++ for Linear search and Binary search (both iterative and recursive versions).

- Change `target` to be searched $n$ times and compute the average empirical time of your algorithm.

- Change length of input data (`array size`) between 10 - 100 and compute the average empirical wall-clock time your algorithm takes to find a `target from the arrays.` **`Hint:`** `Use “random number generator” to` generate array of variable length.