TEMASEK JUNIOR COLLEGE

Temasek Junior College

JC H2 Computing

Problem Solving & Algorithm Design 15
Introduction to Searching algorithms

Searching

1 Searching an array

A common operation on arrays is to search the elements of an array for a particular data item.

The reasons for searching an array may be:

- to edit an input value, i.e. to check that it is a valid element of an array;
- to retrieve information from an array
- to retrieve information from a corresponding element in a paired array.

When searching an array, it is an advantage to have the array sorted into ascending sequence, so that, when a match is found, the rest of the array does not have to be searched. If an array element to be found that is equal to an input entry, a match has been found and the search can be stopped. Also, if an array element to be found that is greater that an input entry, no match has been found and the search can be stopped. Note that if the larger entries of an array are searched more often that the smaller entries, it may be an advantage to sort the array into descending sequence.

An array can be searched using either a linear search or a binary search.

2 A linear/sequential/Serial search of an array

A linear search involves looking at each of the elements of the array, one by one, starting with the first element. Continue the search until either you find the element being looked for or you reach the end of the array.

A linear search is often used to validate data items.

1. Read the search key from user.
2. Compare search key with the start element of the array.
3. If **both matching**, return element **found** and **terminate** search.
4. If both **not matching**, move on to the next element and repeat step 3 and 4 until last element is compared.
5. If **search key and last element still not matching**, return element **not found** and **terminate** search.

Arr	56	72	12	77	23	11	28	45	80	5
index	1	2	3	4	5	6	7	8	9	10

input_value [23] 13

2.1 Linear Search Algorithm

```
Linear_Search_of_an_array(Arr, input_value)

   Set max_num_elements to required value

   Set element_found to false
   //This flag, initially set to false, will be set to true once the
   //value being looked for is found, that is, when the current
   //element of the array is equal to the data item being looked for.

   Set index to 1  // start element of the array
   DOWHILE (NOT element_found) AND (index <= max_num_elements)

         IF Arr(index) = input_value THEN
                Set element_found to true
         ELSE
            index ← index + 1
         ENDIF

   ENDDO

   IF element_found = true THEN
          Print "Element ", Arr(index), " found in ", index
   ELSE
          Print input_value, " not found."
   ENDIF
END
```

Arr	56	72	12	77	23	11	28	45	80	5
index	1	2	3	4	5	6	7	8	9	10

input_value [23] 13

Example 1

Write a program SH1_LinearSearching.py looking for a value in a not in order array of values.

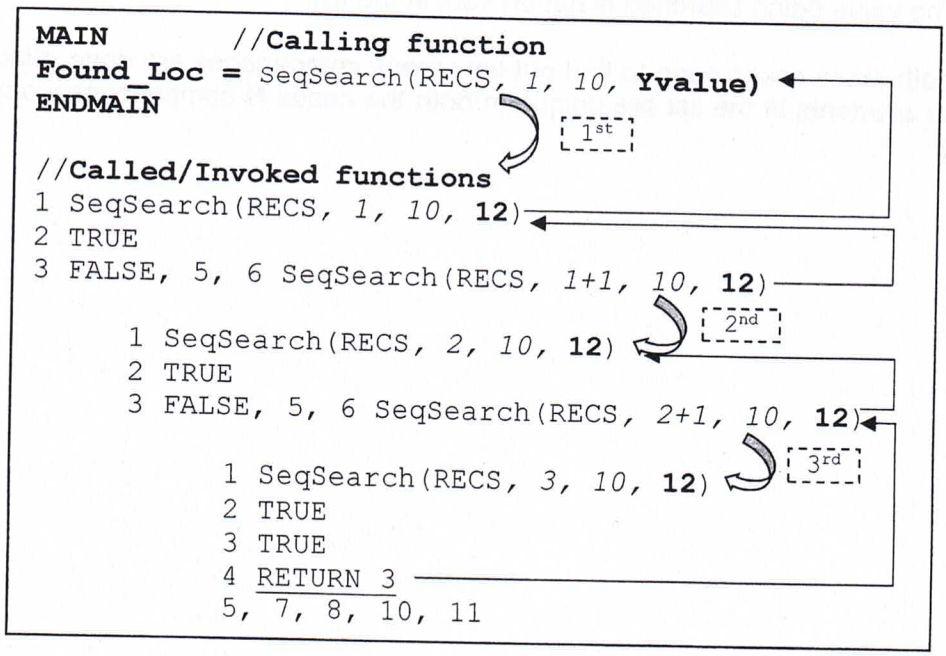
2.2 Recursive Algorithm for Linear/Sequential Search

Algorithm:

```
1 Function SeqSearch(L, I, J, X)

      Input: L is an array, I and J are positive integers, I ≤ J,
      and X is the key to be searched for in L.

      Output: If X is in L between indexes I and J, then output its
      index, else output 0.

2      IF I <= J THEN                    //Base case or Stop condition.
3           IF L(I) = X THEN
4                RETURN I
5           ELSE
6                RETURN SeqSearch(L, I+1, J, X)
7           ENDIF
8      ELSE
9           RETURN 0
10     ENDIF
11 ENDSeqSearch
```

```
MAIN
      DECLARE RECS AS ARRAY[1..10]: INTEGER
      DECLARE Found_Loc: INTEGER
      PROMPT AND GET Yvalue        // 12
      Found_Loc ← SeqSearch(RECS, 1, 10, Yvalue)
      IF Found_Loc = 0 THEN
          PRINT "Not found. "
      ELSE
          PRINT "Yvalue is in the ", Found_Loc, " record. "
      ENDIF
ENDMAIN
```

```
RECS ← [3, 15, 12, 4, 10, 6, 23, 17, 34, 2]
```



Example 2

Write a program SH2_LinearSearchingRecursive.py looking for a value an array of values.

2.3 Efficiency of Linear/Sequential Search

For the linear search algorithm, the number of steps depends on whether the target is in the list, and if so, where in the list, as well as on the length of the list.

For search algorithms, the main steps are the comparisons of list values with the target value.

Counting these for data models representing the **best case**, the **worst case**, and the **average case** produces the following table. For each case, the number of steps is expressed in terms of N, the number of items in the list.

Model	Number of Comparisons (for N = 100000)	Comparisons as a function of N
Best Case (fewest comparisons)	1 (target is first item)	1
Worst Case (most comparisons)	100000 (target is last item)	N
Average Case (average number of comparisons)	50000 (target is middle item)	N/2

2.3.1 Best Case Analysis

The **best case analysis doesn't tell us much**. If the first element checked happens to be the target, *any* algorithm will take only one comparison. O(1).

2.3.2 Worst Case Analysis

There are two worst cases for the sequential search algorithm:

- The value being searched matches the last element in the list
- The value being searched is not present in the list.

For both these cases need to find out how many comparisons are done. Since assumed all the elements in the list are unique in both the cases N comparisons would be made. O(N).

2.3.3 Average Case Analysis

There are two average-case analyses that can be done for a sequential search algorithm. The first assumes that the search is always successful and the other assumes that the value being searched will sometimes not be found.

If the value begin searched is present in the list, it can be present at any one of the N places. Since all these possibilities are equally likely, there is a probability of 1/N for each potential location.

The number of comparisons made if the value being searched is found at first, second, third location, etc. would be 1, 2, 3, and so on. This means that the number of comparisons made is the same as the location where the match occurs. This gives the following equation for this average case:

$$T(N) = \frac{1}{N}\left[\sum_{i=1}^{N} i\right]$$

$$T(N) = \frac{1}{N}\left[\frac{N(N+1)}{2}\right] = \frac{N+1}{2}$$

Now consider the possibility that the value being searched is not present in the list. Now there are N+1 possibilities. For the case where the value being searched is not in the list there will be N comparisons. If we assume that all N+1 possibilities are equally likely, the following equation gives this average case:

$$T(N) = \left[\frac{1}{N+1}\right] * \left[\left[\sum_{i=1}^{N} i\right] + N\right]$$

$$T(N) = \left[\frac{1}{N+1} * \sum_{i=1}^{N} i\right] + \left[\frac{1}{N+1} * N\right]$$

$$T(N) = \left[\frac{1}{N+1} * \frac{N(N+1)}{2}\right] + \frac{N}{N+1}$$

$$T(N) = \frac{N}{2} + \frac{N}{N+1}$$

$$T(N) = \frac{N}{2} + 1 - \frac{1}{N+1}$$

$$T(N) \approx \frac{N+2}{2}$$ (As N gets very large, $\frac{1}{N+1}$ becomes almost 0)

Observe that by including the possibility of the target not being in the list only increases the average case by ½. When consider this amount relative to the size of the list, which could be very large, this ½ is not significant. O(N)

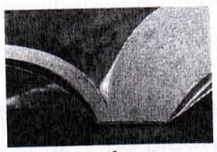
- Sequential search is therefore an **O(N)** algorithm in both the worst case and the average case.

3 A Binary Search of an array

A binary search locates the middle element of the array first, and determines if the element being searched for is in the first half or second half of the table. The search then points to the middle element of the relevant half table, and the comparison is repeated. This technique of continually halving the area under consideration is continued until the data item being searched for is found, or its absence is detected.

1/2

1/4



1/8

1/16 .....

3.1 Binary_Search_of_an_array

```
Set element_found to false
Set low_element to 1
Set high_element to max_num_elements
   DOWHILE (NOT element_found) AND (low_element <= high_element)
      middle ← (low_element + high_element)/2          //middle
      IF array(middle) = input_value THEN
            Set element_found to true
      ELSE
            IF input_value < array(middle) THEN
                  high_element ← middle - 1
            ELSE
                  low_element ← middle + 1
            ENDIF
      ENDIF
   ENDDO
   IF element_found = true THEN
            Print "Element ", array(middle), " found in ", middle
   ELSE
            Print input_value, " not found."
   ENDIF
END
```

Testing data: 52, 45

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
data	23	27	29	32	34	41	46	47	49	52	55	68	71	74	77	78
1st iteration	low							middle								high
2nd iteration									low			middle				high
3rd iteration									low	middle	high					
	low							middle								high
	low			middle			high									
					low	middle	high									
							low high middle									
						high	low									

Example 3

Write a program SH3_BinarySearching.py looking for a value an array.

3.2 Binary search using recursion

The binary search algorithm is a method of searching an ordered array for a single element by cutting the array in half with each pass.

The trick is to pick a midpoint near the center of the array, compare the data at that point with the data being searched and then responding to one of three possible conditions:

- the data is found at the midpoint,
- the data at the midpoint is greater than the data being searched for,
- or the data at the midpoint is less than the data being searched for.

Recursion is used in this algorithm because with each pass a new array is created by cutting the old one in half. The binary search procedure is then called recursively, this time on the new (and smaller) array. Typically the array's size is adjusted by manipulating a beginning and ending index. The algorithm exhibits a logarithmic order of growth because it essentially divides the problem domain in half with each pass.

```
//  Call binary_search with proper initial conditions.
//  INPUT:
//    data is an array of integers SORTED in ASCENDING order,
//    toFind is the integer to search for,
//    count is the total number of elements in the array
//
//  OUTPUT:
//    result of binary_search

search(data, toFind, count)
    Start ← 0          //beginning index
    End ← count - 1  //top index
    return Recursive_binary_search(data, toFind, 0, count-1)
ENDsearch
//   Binary Search Algorithm.
//  INPUT:
//        data is a array of integers SORTED in ASCENDING order,
//        toFind is the integer to search for,
//        start is the minimum array index,
//        end is the maximum array index
//   OUTPUT:
//  position of the integer toFind within array data, -1 if not found
Recursive_binary_search(data, toFind, start, end)
//Get the midpoint.
    mid ← (start + end)/2    //Integer division

    IF (start > end) THEN//Not found -Base case or Stop condition.
              return -1
    ELSEIF (data[mid] = toFind) //Found
        return mid
    ELSEIF (data[mid] > toFind) //Data is greater than toFind, search lower half
       return Recursive_binary_search(data, toFind, start, mid-1)
    ELSE                        //Data is less than toFind, search upper half
        return Recursive_binary_search(data, toFind, mid+1, end)
    ENDIF
ENDRecursive_binary_search
```

Example 4

Write a program SH4_BinarySearchingRecursive.py looking for a value an array.

3.3 Efficiency of Binary Search

To evaluate binary search, count the number of comparisons in the **best case and worst case**. This analysis **omits the average case**, which is a bit more difficult, and ignores any differences between algorithms in the amount of computation corresponding to each comparison.

3.3.1 Best Case

The best case occurs if the middle item happens to be the target. Then only one comparison is needed to find it. As before, the best case analysis does not reveal much.

3.3.2 Worst Case

If the target is not in the array then the process of dividing the list in half continues until there is only one item left to check. Here is the pattern of the number of comparisons after each division, given the simplifying assumptions of an initial array length that's an even power of 2 (1024) and exact division in half on each iteration:

Items Left to Search	Comparisons So Far
1024	0
512	1
256	2
128	3
64	4
32	5
16	6
8	7
4	8
2	9
1	10

For a list size of 1024, there are 10 comparisons to reach a list of size one, given that there is one comparison for each division, and each division splits the list size in half.

In general, if n is the size of the list to be searched and C is the number of comparisons to do so in the worst case, $\mathbf{C = log_2 N}$. Thus, the efficiency of binary search can be expressed as a logarithmic function, in which the number of comparisons required to find a target increases only logarithmically with the size of the list.

Binary Search does $O(\log_2 N)$ comparisons in the worst case.

The following table summarizes the analysis for binary search.

Model	Number of Comparisons (for N = 100000)	Comparisons as a function of N
Best Case (fewest comparisons)	1 (target is middle item)	1
Worst Case (most comparisons)	16 (target not in array)	$\log_2 N$

3.3.3 Average Case

As a matter of fact, it also does comparisons in the average case to find a name that is in the list (although the exact value is a smaller number than in the worst case).

The average cost of a successful search is about the same as the worst case where an item is not found in the array, both being roughly equal to $\log_2 N$.

3.4 Efficiency Comparison

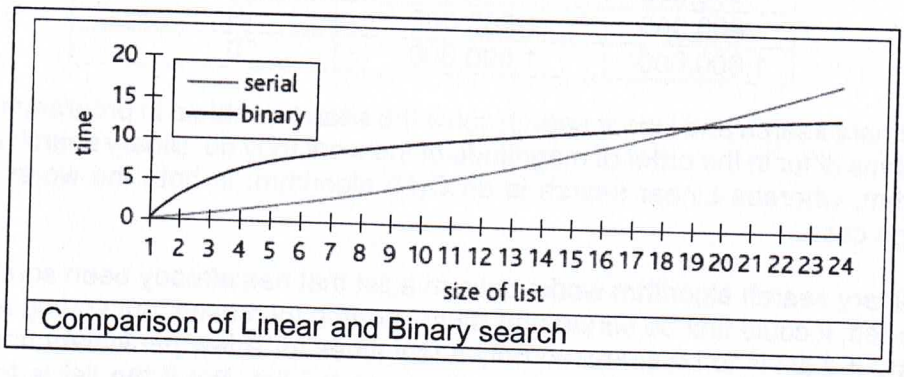
In order to compare the efficiency of the two methods, lists of different lengths will be used.

To do this we need to know the number of comparisons that need to be made for lists of different lengths.

As a binary search halves the lists each time we do a comparison, the number of comparisons is given by m where 2m is the smallest value greater than or equal to n, the number of values in the list.

Suppose there are 16 values in a list. The successive lists compared contain 16, 8, 4 and 2 values. This involves a maximum of four comparisons and 24 is 16. Similarly, we only need 10 comparisons if the list contains 1000 values because 210 is equal to 1024, which is close to and greater than 1000.

Figure below shows a comparison of the search times for the two methods. It includes the calculation of the midpoints in the lists used by the binary search and the time taken to make comparisons. It does not **include sorting time before a binary search.**



Comparison of Linear and Binary search

	Comparison of serial and binary search methods.	
	Advantages	**Disadvantages**
Serial search	• Additional data are simply appended to the end of the list. • The search algorithm is straightforward. • A search works well if there is a small number of data items.	• A search is inefficient if there is a large number of data items. • Missing data items cannot be identified until every item has been compared.
Binary search	• Large numbers of data items can be searched very quickly. • Missing data items can be identified quickly.	• Data items must be sorted before the search is carried out. • Adding or deleting a data item is slow (as every change involves re-sorting the data).

Note that the worst case number of comparisons is just 16 with 100,000 items, versus 100,000 for linear search! Furthermore, if the array were doubled in size to 200,000, the maximum number of comparisons for binary search would only increase by 1 to 17, whereas for linear search it would double from 100,000 to 200,000.

The following table shows how the maximum number of comparisons increases for binary search and linear search.

Array Size	**Worst Case Comparisons**	
	Linear Search	**Binary Search**
100,000	100,000	16
200,000	200,000	17
400,000	400,000	18
800,000	800,000	19
1,600,000	1,600,000	20

- Both Binary search and Linear search solve the search problem in programming, but these algorithms differ in the order of magnitude of the work they do. Binary search is an $O(\log_2 N)$ algorithm, whereas Linear search is an $O(N)$ algorithm, in both the worst case and the average case.

- The Binary search algorithm works only on a list that has already been sorted. If the list is not sorted, it could first be sorted and Binary search then used, but sorting also takes a lot of work. If a list is to be searched only a few times for a few particulars names, then it is more efficient to do Linear search on the unsorted list. But if the list is to be searched repeatedly, it is more efficient to sort it and then use Binary search.

Tutorial 15 [Linear and Binary Searching Algorithms]

1* Linear search and binary search are two different algorithms which can be used for searching arrays. [H446/02 Algo and programming

When comparing linear and binary search it is possible to look at the best, worst and average number of items in the array that need to be checked to find the item being searched for.

Assume every item in the array is equally likely to be searched for.

(a) Copy and complete the table below [4]

	Worst Case number of searches	Average Case	Best Case
Binary Search		$\log_2(N) - 1$	
Linear Search		N/2	

(b) As the size of an array increases the average number of checks grows logarithmically. State what is meant by logarithmic growth.

(c) Assuming an array is sorted give a situation when a linear search would perform better than a binary search. [1]

2* Explain how the two main search algorithms work: linear and binary search.

3 Explain the circumstances where you might use a binary search compared to a linear search.

4* Why is a binary search considered to be more efficient than a linear search on large datasets?

5 Two methods for searching a dataset are a binary search and a linear search.
(a) Write pseudo-code to show how a binary search works.
(b) Explain how efficient this method is on a large ordered set of data, compared to a linear search.

6 Write pseudo code for a binary search and a linear search. Explain the time complexity of each with reference to your code.

7 The common orders of time complexity are shown in the table.

Time complexity
O(1)
$O(n^2)$
$O(\log_2 n)$
$O(k^n)$
O(n)

(a) Describe in words what O(1) means.
(b) What is meant by an intractable problem? [Intractable problem: a problem that cannot be solved within an acceptable time frame.]
(c) Which is the time complexity of an intractable problem?
(d) Which is the time complexity for a binary search?
(e) Which is the time complexity for a linear search?
(f) On average, would a binary search or a linear search be quickest on a list of just five items? Explain your answer.

[Total: 15]

8*

(a) Copy and complete the algorithm, written in pseudocode, for a binary search. The data being searched is stored in the array SearchData[63]. The item of data being searched is stored in the variable SearchItem.

```
X ← 0
Low ← 1
High□.....
WHILE (High>=Low) AND (.....)
    Middle ← INT((High + Low)/2)
    IF SearchData[Middle] = SearchItem
        THEN
            X ← Middle
        ELSE
            IF SearchData[Middle] < SearchItem
                THEN
                    Low ← Middle + 1
                ELSE
                    IF SearchData[Middle] > SearchItem
                        THEN
                        .....
                    ENDIF
            ENDIF
    ENDIF
ENDWHILE
```

[3]

(b)

(i) The binary search only works if the data in the array being searched is: [1]

(ii) The maximum number of comparisons that are required to find an item which is present in the array SearchData is: [1]

(iii) At the end of the algorithm, the variable X contains:

either the value _____ which indicates _____
or the value _____ which indicates _____ [4]

(c) You will change the binary search algorithm to a recursive algorithm and write the equivalent program code in the form of a procedure. Name the recursive procedure BinarySearch.

Use these variables.

Variable	Data Type	Description
SearchData	ARRAY[63] : INTEGER	global array
SearchItem	INTEGER	global variable
X	INTEGER	global variable
Low	INTEGER	parameter
High	INTEGER	global variable
Middle	INTEGER	local variable

Write pseudocode for the recursive procedure BinarySearch. [5]

(d) Write the initial call to the recursive procedure. [1]

BinarySearch(1,63) [1]