Computer Programming (b) - E1124

(Spring 2021-2022)

Lecture 3

Searching Algorithms

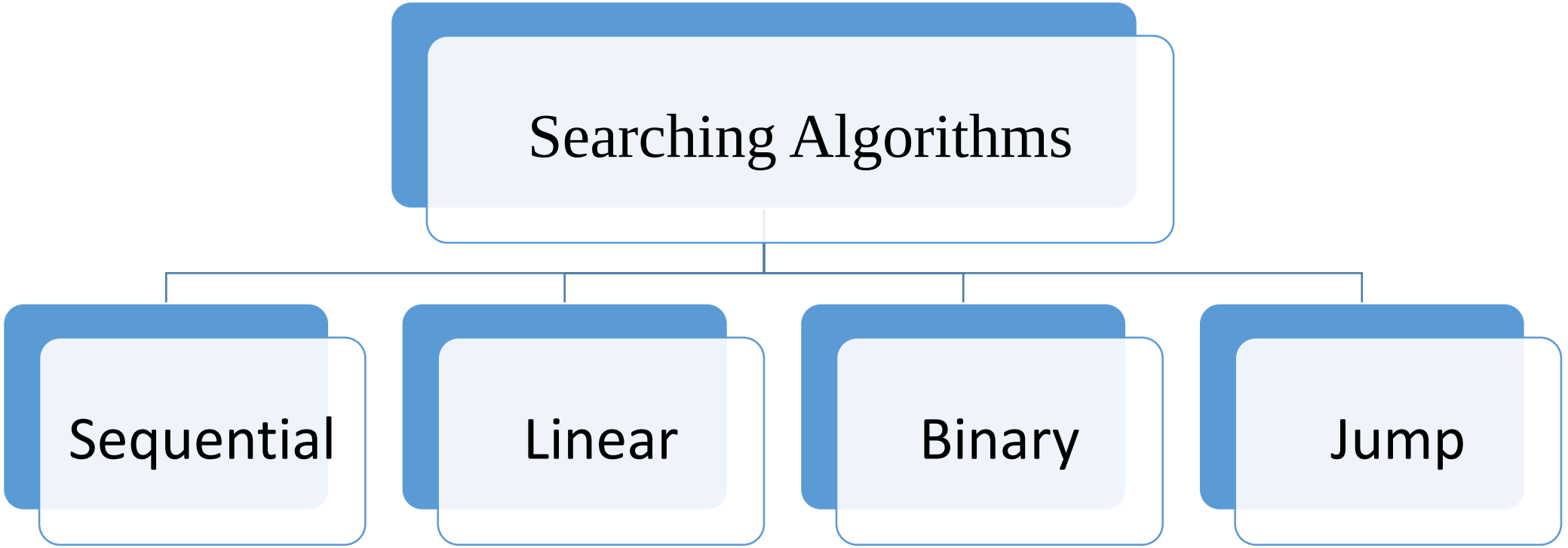
INSTRUCTOR

Dr / Ayman Soliman

Contents

- Sequential Search
- Linear Search
- Binary Search
- Jump Search
- Sorting a List: Bubble Sort
- Sorting a List: Selection Sort
- Sorting a List: Insertion Sort

- **A search algorithm** is the step-by-step procedure used to locate an item within a list of information.



Sequential Search

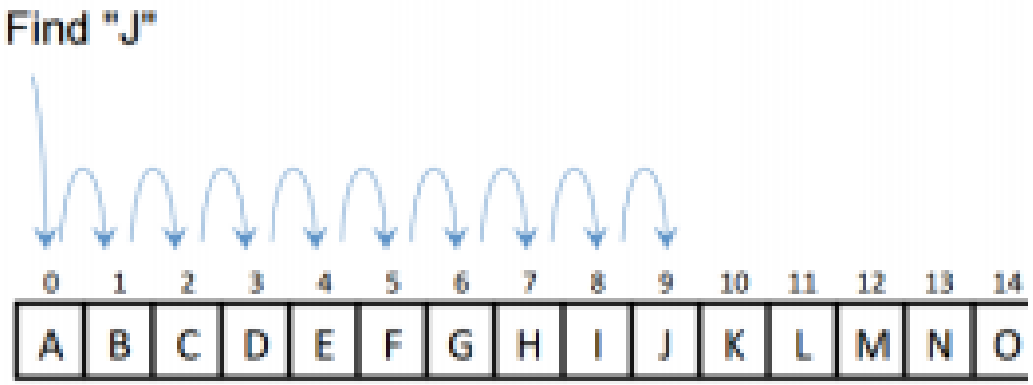
➢ Sequential Search (cont.)

```
int seqSearch(const int list[], int listLength, int searchItem)
{
    int loc;
    bool found = false;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
        {
            found = true;
            break;
        }

    if (found)
        return loc;
    else
        return -1;
}
```

Linear Search

➢ Linear search (sequential search)

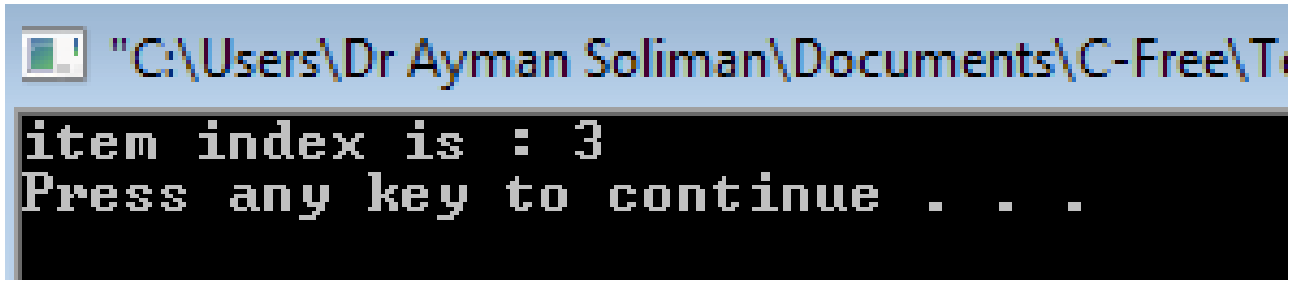
- ➢ It sequentially checks each element starting at the first element of the list for the target value until a match is found or all the elements have been checked.
- ➢ Advantages
  - ✓ Straightforward algorithm.
  - ✓ Array could be in any order.



- ➢ Disadvantages
  - ✓ Time taken to search elements keep increasing as the number of elements are increased.

➢ Example 1

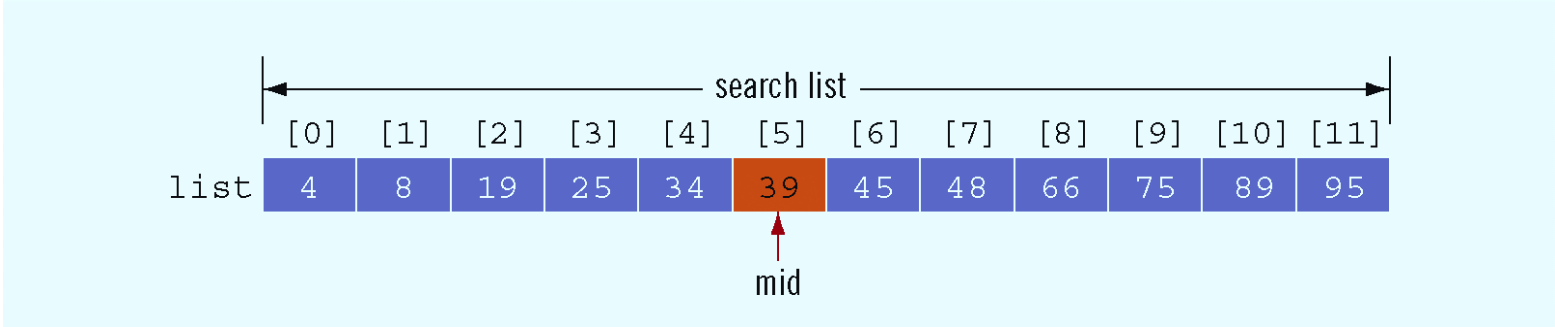
```
1  #include <iostream.h>
2  int seqsearch(int[],int,int);
3  int main(int argc, char *argv[])
4  {
5      int arr[]={1,5,13,4,25};
6      int n=sizeof(arr)/sizeof(arr[0]);
7      cout<<"item index is : "<<seqsearch(arr,n,4)<<endl;
8      return 0;
9  }
10
11 int seqsearch(int arr[],int n,int y)
12 {
13     for(int x=0; x<n; x++)
14     if (arr[x]==y)
15     return x;
16     return -1;
17 }
```



Binary Search

➢ Binary Search (half-interval-search)

- ➢ Binary search can be applied to sorted lists
- ➢ Uses the "**divide and conquer**" technique
  - ❑ Compare search item to middle element
  - ❑ If search item is less than middle element, restrict the search to the lower half of the list
  - ❑ Otherwise search the upper half of the list



➢ Binary Search (cont.)

➢ Advantage
- More efficient than linear search and has time complexity

➢ Disadvantage
- Requires a sorted array.

➢ Binary Search (half-interval-search)

Search for a given number x=100

{ 1, 2, 20, 30, 50, 100, 300}

Check the middle element

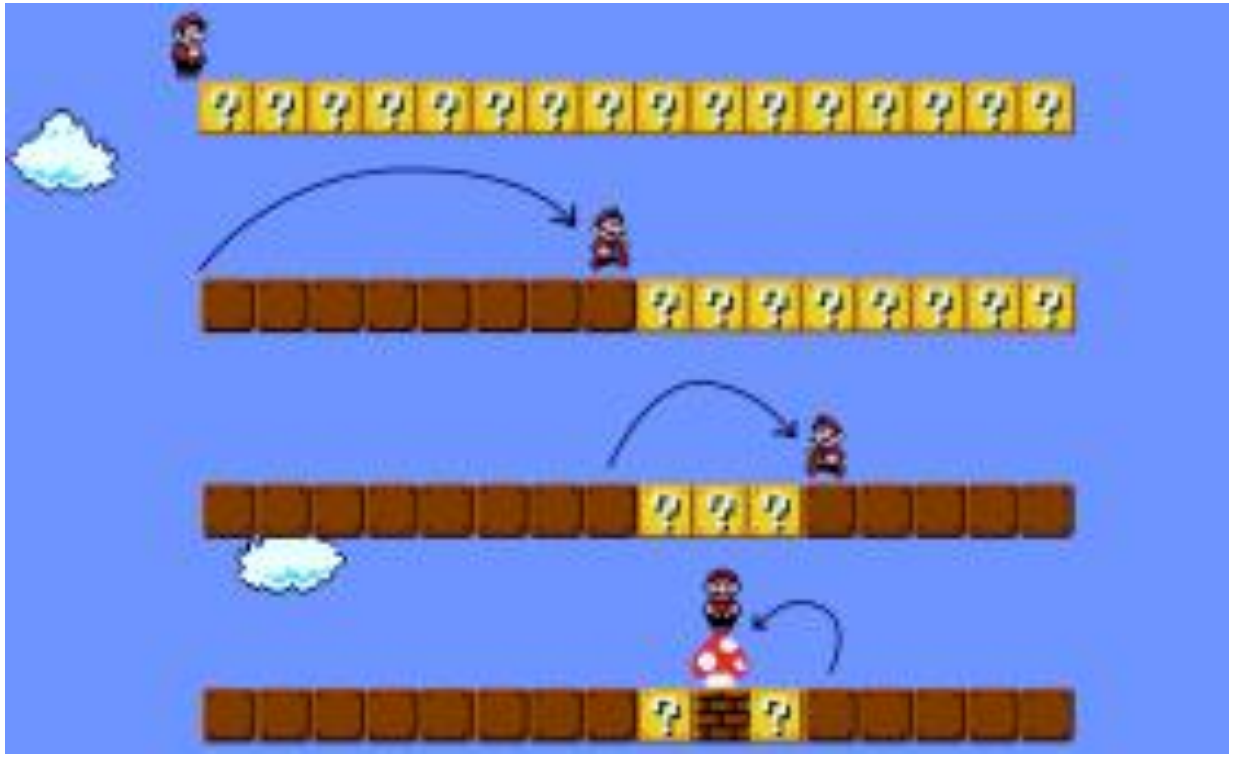
{ 1, 2, 20, 30, 50, 100, 300} //x>30

Check the Right side array

{1, 2, 20, 30, 50, 100, 300}

Check the new middle element

{1, 2, 20, 30, 50, 100, 300} //done



➢ Binary Search

$$mid = \frac{first + last}{2}$$

```
int binarySearch(const int list[], int listLength, int searchItem)
{
    int first = 0;
    int last = listLength - 1;
    int mid;

    bool found = false;

    while (first <= last && !found)
    {
        mid = (first + last) / 2;

        if (list[mid] == searchItem)
            found = true;
        else if (list[mid] > searchItem)
            last = mid - 1;
        else
            first = mid + 1;
    }

    if (found)
        return mid;
    else
        return -1;
}//end binarySearch
```

➢ Example 2

➢ Search item = 89

Iteration	first	last	mid	list[mid]	No. of key comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1 (found is true)

➢ Example 3

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

➢ Search item = 22

Iteration	first	last	mid	list[mid]	No. of key comparisons
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	3	2	the loop stops (since first > last) unsuccessful search		

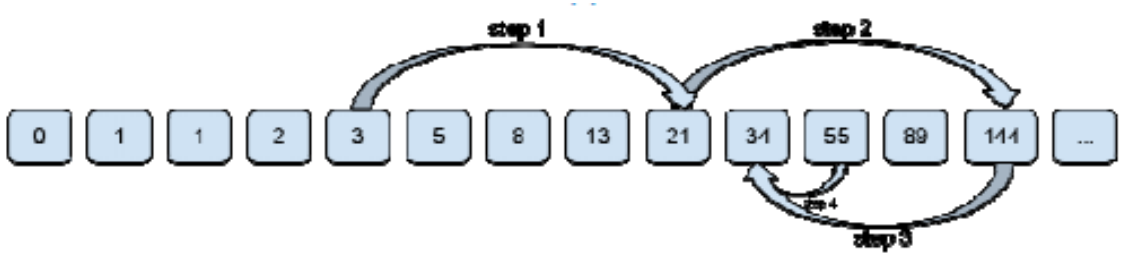
Binary Search (cont.)

- Every iteration cuts size of search list in half
- If list L has 1000 items
  - At most 11 iterations needed to determine if an item x is in list
- Every iteration makes 2 key (item) comparisons
  - Binary search makes at most 22 key comparisons to determine if x is in L
- Sequential search makes 500 key comparisons (average) to if x is in L for the same size list

Jump Search

Jump search

- **Jump Search** is used for sorted arrays by jumping ahead by fixed steps (m) or skipping some elements instead of searching all elements and once we find the interval, we perform a linear search operation till finding the search key.



- Search for a given number x=55
  { 0,1, 1, 2, 3, 5, 8, 13, 21, 31, 55, 89, 144, 150, 160, 170} // length=16
  Jump by fixed step m=4
  { 0,1, 1, 2, 3, 5, 8, 13, 21, 31, 55, 89,144, 150, 160, 170}
  Once we find the interval.
  { 0,1, 1, 2, 3, 5, 8, 13, 21, 31, 55, 89,144, 150, 160, 170} //21<x<144
  Perform a linear search
  {34, 55, 89}

➢ Jump search (cont.)

- ➢ Advantage
  - • More efficient than linear search and has time complexity

- ➢ Disadvantage
  - • Requires a sorted array.

➢ Jump search (cont.)

The search key is x=30

{ 1, 2, 20, 30, 100, 300} → L = 0, r = $m = \sqrt{n} = \sqrt{6} = 2\ (integer)$

{ 1, 2, 20, 30, 100, 300} → L = 0 , r = m = 2, x > 20

{ 1, 2, 20, 30, 100, 300} → L = r = 2 , r = r + m = 4, x< 100

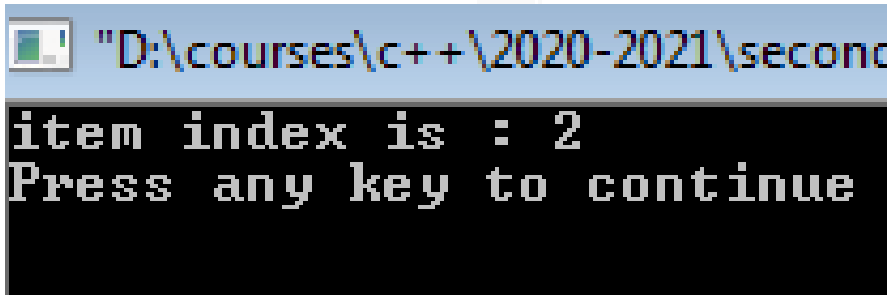
{ 1, 2, 20, 30, 100, 300} → Linear search starting from 20 → 100, or starting from 30 if you already checked indices at boundaries.

{ 1, 2, 20, 30, 100, 300} → L= 30

➢ Example 4

```
#include <iostream>
#include <cmath>
using namespace std;
int jumpsearch(int[],int,int);
int main(int argc, char *argv[])
{
    int arr[]={1,5,13,4,25};
    int n=sizeof(arr)/sizeof(arr[0]);
    int x=13;
    int index=jumpsearch(arr,n,x);
    cout<<"item index is : "<<index<<endl;
    return 0;
}
```

```
int jumpsearch(int arr[],int n,int x)
{
    int l=0, r=sqrt(n), m=sqrt(n);
    while(arr[r]<=x && r<n)
    {   l=r;
        r=r+m;
        if (r>n-1)
        r=n;}
    for (int i=l;i<r; i++)
    if (arr[i]==x)
    return i;
    return -1;}
```



Thank you