Chapter Two

Simple Sorting & Searching Algorithms

Ashenafi C.

Addis Ababa Science and Technology University (AASTU)

Sorting Algorithm

- Sorting is a technique to rearrange the elements of a list in ascending or descending order.
- A *sorting algorithm* is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order.
- **Sorting Example:**
- Given a set (container) of n elements
  - E.g. array, set of words, etc.
- Suppose there is an order relation that can be set across the elements
- Goal Arrange the elements in ascending order

  - Start → 1 23 2 56 9 8 10 100
  - End → 1 2 8 9 10 23 56 100

Classification

- Sorting can be classified in two types:
- **Internal Sorting:-**
  - Uses only the primary memory during sorting process.
  - All data items are held in main memory and no secondary memory is required
  - **Limitation**: they can only process relatively small lists due to memory constraints. There are 3 types of internal sorts.
  - E.g. selection sort, insertion sort, bubble sort

Classification

- **External Sorting:-**
  - Sorting large amount of data requires external or secondary memory.
  - This process uses external memory such as HDD, to store the data which is not fit into the main memory.
  - So, primary memory holds the currently being sorted data only.
  - All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.
  - **Ex:- Merge Sort**

Popular Sorting Algorithms

- While there are a large number of sorting algorithms, in practical implementations a few algorithms predominate:
  - Insertion sort
  - Bubble sort
  - Selection sort

Simple

  - Merge sort
  - Quick sort
  - Heap sort

Advanced

Insertion Sort: Idea

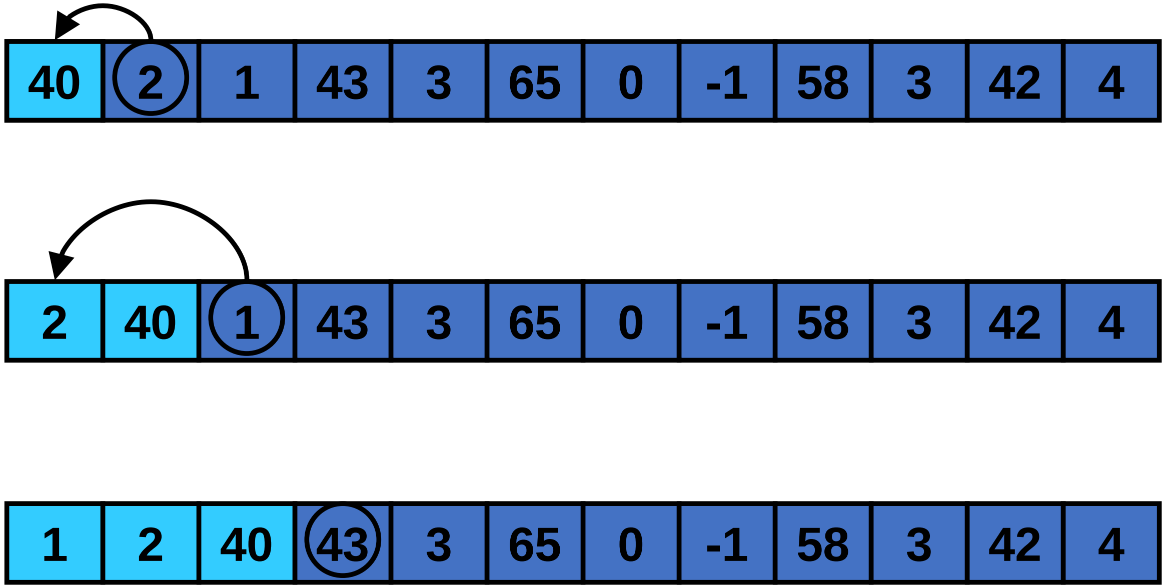
1. Create two group of items:
   - sorted group, and
   - unsorted group
2. We assume that items in the unsorted group are unsorted, and we have to keep items in the sorted group sorted.
3. Pick any item from unsorted, then insert the item at the right position in the sorted group to maintain sorted property.
4. Repeat the process until the unsorted group becomes empty.

Insertion Sort process:

- 1. The left most value can be said to be sorted relative to itself. Thus, we don‘t need to do anything.
- 2. Check to see if the second value is smaller than the first one. If it is, swap these two values. The first two values are now relatively sorted.
- 3. Next, we need to insert the third value in to the relatively sorted portion so that after insertion, the portion will still be relatively sorted.
- 4. Remove the third value first. Slide the second value to make room for insertion. Insert the value in the appropriate position.
- 5. Now the first three are relatively sorted.
- 6. Do the same for the remaining items in the list.

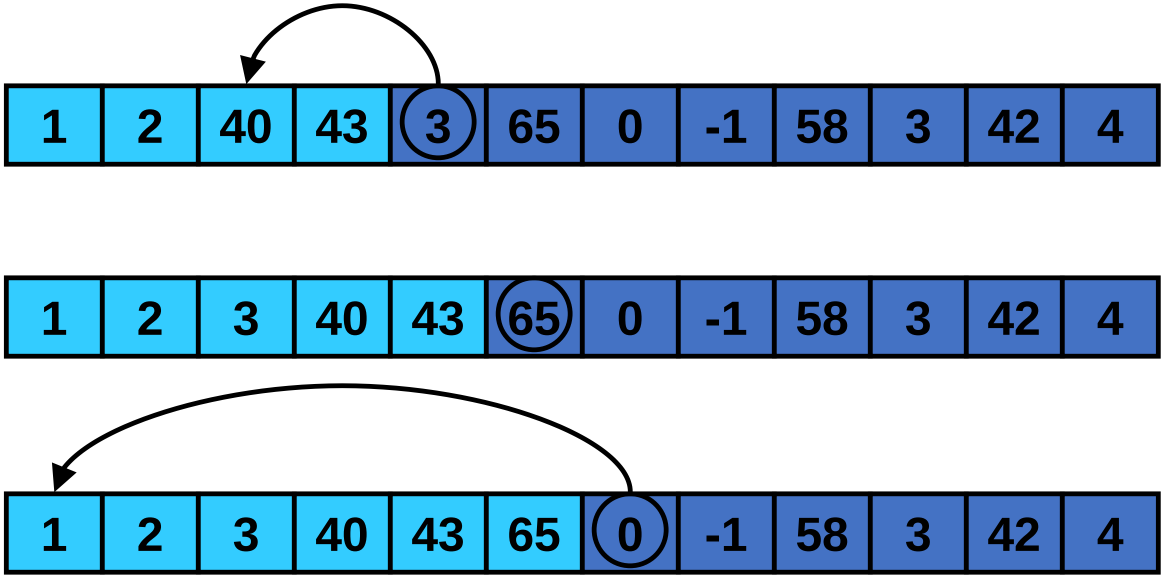
- **Sorting cards:**
  - 8 | 5 9 2 6 3
  - 5 8 | 9 2 6 3
  - 5 8 9 | 2 6 3
  - 2 5 8 9 | 6 3
  - 2 5 6 8 9 | 3
  - 2 3 5 6 8 9 |

Insertion Sort: Example



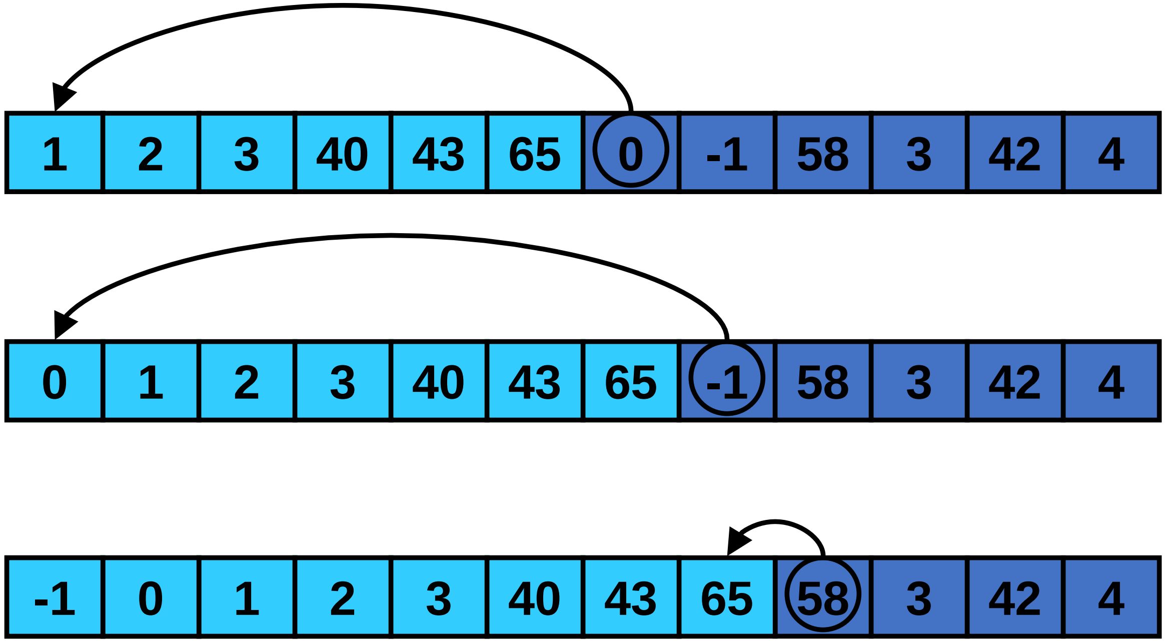
40	2	1	43	3	65	0	-1	58	3	42	4
2	40	1	43	3	65	0	-1	58	3	42	4
1	2	40	43	3	65	0	-1	58	3	42	4

Insertion Sort: Example



1	2	40	43	3	65	0	-1	58	3	42	4
1	2	3	40	43	65	0	-1	58	3	42	4
1	2	3	40	43	65	0	-1	58	3	42	4

Insertion Sort: Example

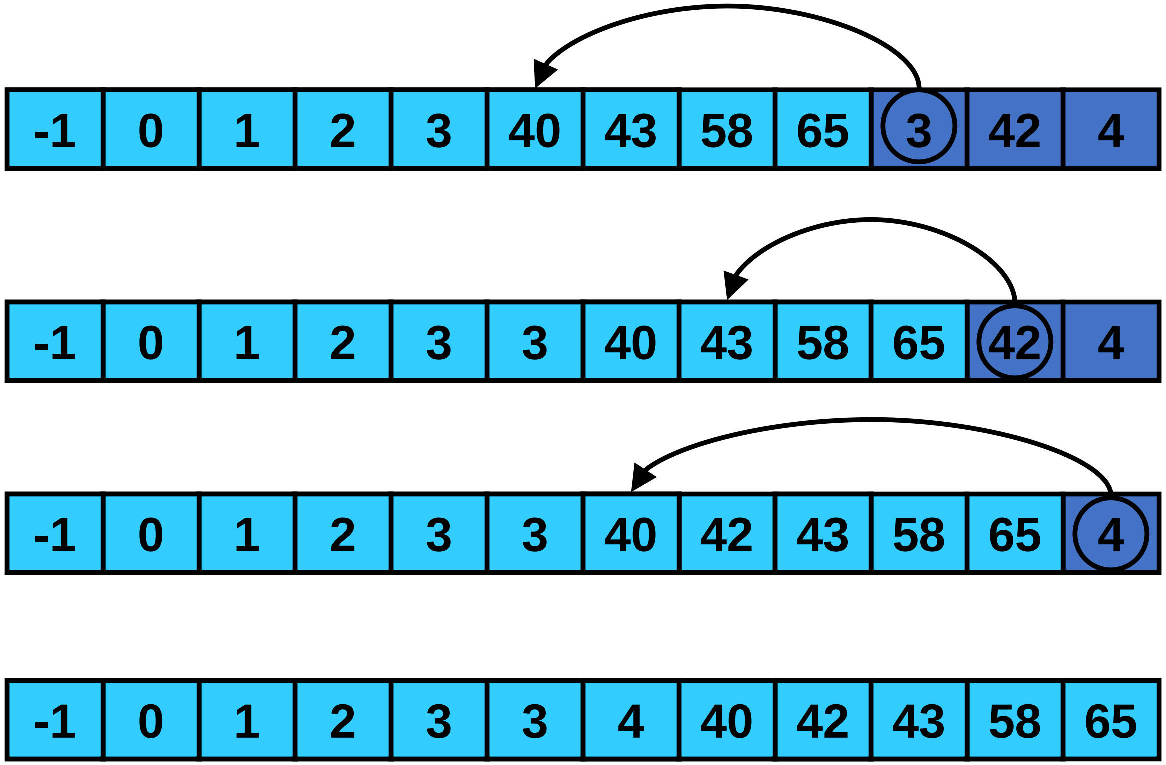


1	2	3	40	43	65	0	-1	58	3	42	4

0	1	2	3	40	43	65	-1	58	3	42	4

-1	0	1	2	3	40	43	65	58	3	42	4

Insertion Sort: Example



-1	0	1	2	3	40	43	58	65	3	42	4

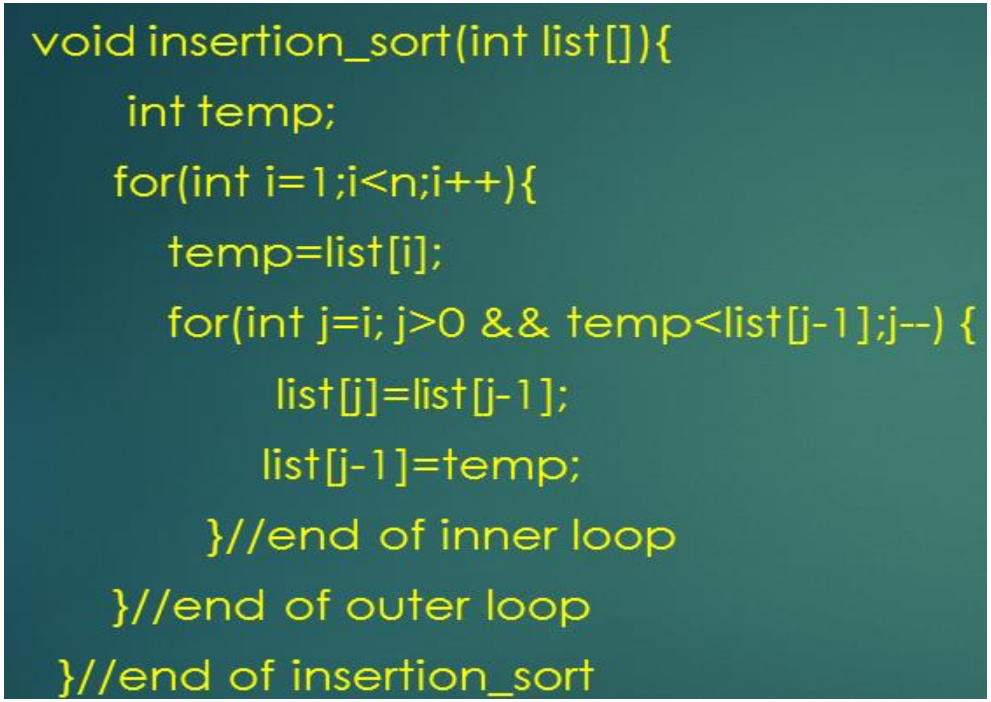
-1	0	1	2	3	3	40	43	58	65	42	4

-1	0	1	2	3	3	40	42	43	58	65	4

-1	0	1	2	3	3	4	40	42	43	58	65

Implementation:

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



```
void insertion_sort(int list[]){
       int temp;
      for(int i=1;i<n;i++){
         temp=list[i];
         for(int j=i; j>0 && temp<list[j-1];j--) {
                 list[j]=list[j-1];
                list[j-1]=temp;
            }//end of inner loop
      }//end of outer loop
  }//end of insertion_sort
```

Analysis

How many comparisons?

$$1+2+3+\ldots+(n-1) = O(n^2)$$

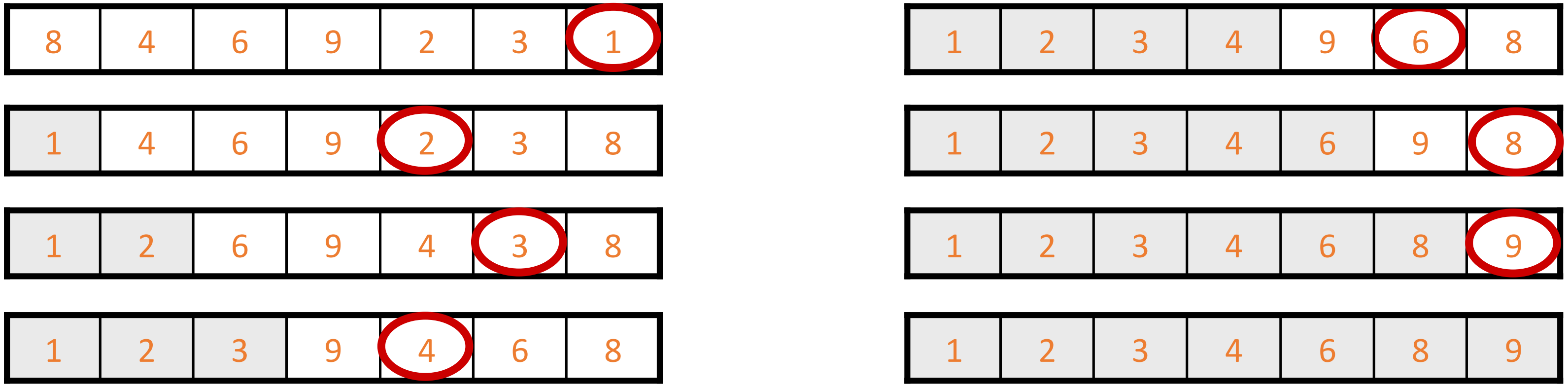
How many swaps?

$$1+2+3+\ldots+(n-1) = O(n^2)$$

Selection Sort: Idea

1. We have two group of items:
   - sorted group, and
   - unsorted group
2. Initially, all items are in the unsorted group. The sorted group is empty.
   - We assume that items in the unsorted group unsorted.
   - We have to keep items in the sorted group sorted.
3. Select the “best” (e.g. Smallest or largest) item from the unsorted group, then put the “best” item at the end of the sorted group.
4. Repeat the process until the unsorted group becomes empty.

Example



8	4	6	9	2	3	1
1	4	6	9	2	3	8
1	2	6	9	4	3	8
1	2	3	9	4	6	8
1	2	3	4	9	6	8
1	2	3	4	6	9	8
1	2	3	4	6	8	9
1	2	3	4	6	8	9

Implementation:

- Loop through the array from i=0 to n-1.
- Select the smallest element in the array from i to n
- Swap this value with value at position i.

```
void selection_sort(int list[]){
  int i,j, smallest;
  for(i=0;i<n;i++){
     smallest=i; //assume ith pos. smallest
     for(j=i+1;j<n;j++){
      if(list[j]<list[smallest])
          smallest=j;
        }//end of inner loop
       temp=list[smallest];
  swap list[smallest]=list[i];
       list[i]=temp;
  } //end of outer loop
}//end of selection_sort
```

Bubble Sort: Definition

- Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.

  - Idea:
    - Repeatedly pass through the array
    - Swaps adjacent elements that are out of order

Bubble Sort

9, 6, 2, 12, 11, 9, 3, 7

Compares the numbers in pairs from left to right exchanging when necessary. The first number is compared to the second and as it is larger they are exchanged.

6, 9, 2, 12, 11, 9, 3, 7

The next pair of numbers are compared. 9 is the larger and this pair is also exchanged.

6, 2, 9, 12, 11, 9, 3, 7

In the third comparison, the 9 is not larger than the 12 so no exchange is made. We move on to compare the next pair without any change to the list.

Bubble sort

6, 2, 9, 12, 11, 9, 3, 7

The 12 is larger than the 11 so they are exchanged.

6, 2, 9, 11, 12, 9, 3, 7

The 12 is greater than the 9 so they are exchanged

6, 2, 9, 11, 9, 12, 3, 7

The 12 is greater than the 3 so they are exchanged.

Bubble sort

6, 2, 9, 11, 9, 3, 12, 7

The 12 is greater than the 7 so they are exchanged.

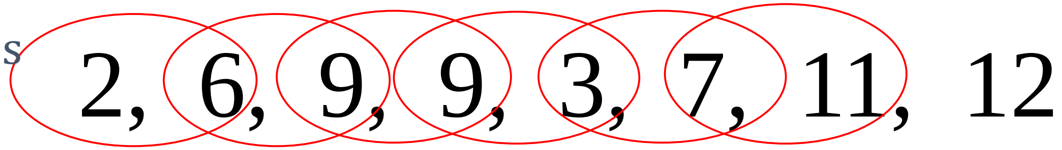
The end of the list has been reached so this is the end of the first pass. The twelve at the end of the list must be largest number in the list and so is now in the correct position. We now start a new pass from left to right.

6, 2, 9, 11, 9, 3, 7, 12

Bubble sort

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass



2, 6, 9, 9, 3, 7, 11, 12

This time we do not have to compare the last two numbers as we know the 12 is in position.
This pass therefore only requires 6 comparisons.

Bubble sort

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Third Pass 2, 6, 9, 3, 7, 9, 11, 12

This time the 11 and 12 are in position. This pass therefore only requires 5 comparisons.

Bubble sort

First Pass	6, 2, 9, 11, 9, 3, 7, 12
Second Pass	2, 6, 9, 9, 3, 7, 11, 12
Third Pass	2, 6, 9, 3, 7, 9, 11, 12
Fourth Pass	2, 6, 3, 7, 9, 9, 11, 12

Each pass requires fewer comparisons. This time only 4 are needed.

Bubble sort

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Third Pass 2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass 2, 6, 3, 7, 9, 9, 11, 12

Fifth Pass 2, 3, 6, 7, 9, 9, 11, 12

The list is now sorted but the algorithm does not know this until it completes a pass with no exchanges.

Bubble sort

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Third Pass 2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass 2, 6, 3, 7, 9, 9, 11, 12

This pass no exchanges are made so the algorithm knows the list is sorted. It can therefore save time by not doing the final pass. With other lists this check could save much more work.

Fifth Pass 2, 3, 6, 7, 9, 9, 11, 12

Sixth Pass 2, 3, 6, 7, 9, 9, 11, 12

Implementation:

```
void bubble_sort(list[]){
  int i,j,temp;
  for(i=0;i<n; i++){
    for(j=0;j<n-i-1; j++){
      if(list[j]>list[j+1]){
        temp=list[j];
        list[j]=list[j+1];
        list[j+1]=temp;
      }//swap adjacent elements
    }//end of inner loop
  }//end of outer loop
}//end of bubble_sort
```

Analysis of Bubble Sort:

- How many comparisons?

  $(n-1)+(n-2)+\ldots+1= O(n^2)$

- How many swaps?

  $(n-1)+(n-2)+\ldots+1= O(n^2)$

- Total $=O(n^2)$

Simple Searching Algorithms

Linear Search

- Algorithm involves checking all the elements of the array (or any other structure) one by one and in sequence until the desired result is found.

Daily life example:

- If you are asked to find the name of the person having phone number say “1234” with the help of a telephone directory.

- Since telephone directory is sorted by name not by numbers, we have to go through each and every number of the directory.

Best case

- If the first number in the directory is the number you were searching for, then lucky you!
- Since you have found it on the very first page, now its not important for you that how many pages are there in the directory.
- Whether if it is of 1000 pages or 2000 pages it will take you same time to find you the number, if it is at the very beginning .
- So it does not depend on the number elements in the directory. **Hence constant time: O(1)**

Worst case

- It may happen that the number you are searching for is the last number of directory or if it is not in the directory at all.
- In that case you have to search the whole directory.

- Now number of elements will matter to you. If there are 500 pages, you have to search 500; if it has 1000 you have to search 1000.

- Your search time is proportional to number of elements in the directory. **Hence: O(n)**

Implementation:

Pseudocode:

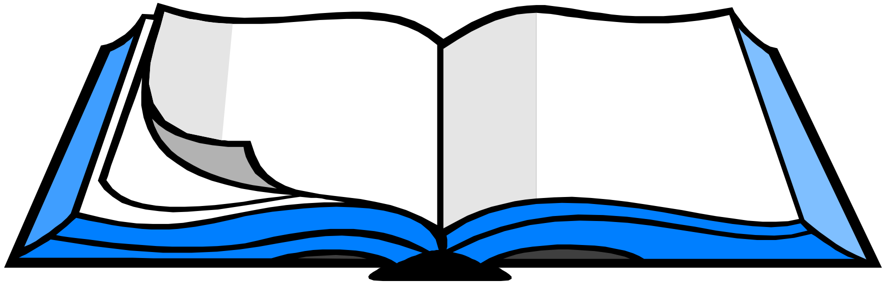
```
For all elements
   Check if it is equal to element being searched for.
   If it is, return its position.
   else continue.
```

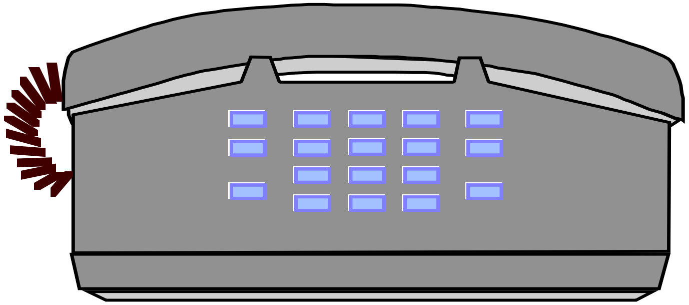
```
//const for safety ,we want to keep array unchanged
void linearSearch(const double data [ ], int n, double key) {

    for(int i=0;i<=n-1;i++) //looping through all elements of the array{
        if(data[i]==key){
            cout<<key<<" found at index "<<i<<" of the array"<<endl;
            break; //if element is found, come out of the loop
         }
      if(i==n-1) //searched through the array, still not found
         cout<<"\n Element not found \n";
    }
}
```

Binary Search

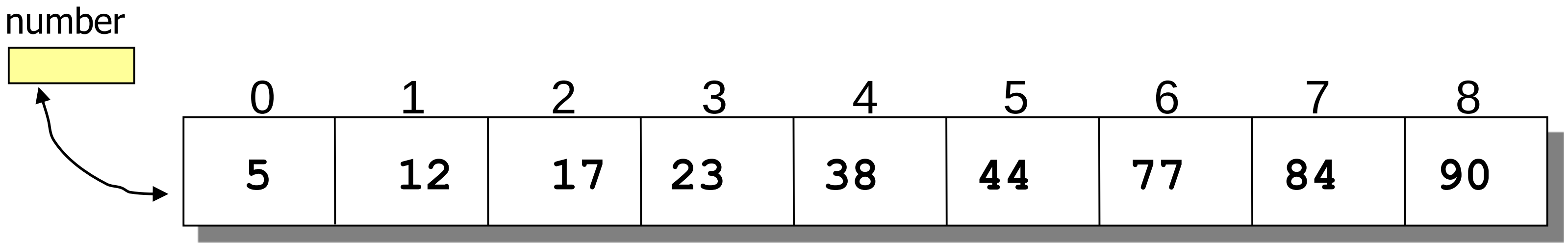
- Binary search is like looking up a phone number or a word in the dictionary.
  - Start in middle of book.
  - If name you're looking for comes before names on page, look in first half.
  - Otherwise, look in second half.





Binary Search

- If the array is **sorted**, then we can apply the binary search technique.



number

0	1	2	3	4	5	6	7	8
5	12	17	23	38	44	77	84	90

- The basic idea is straightforward. First search the value in the middle position. If value is middle element, lucky you!
- If X is less than this value, then search the middle of the left half next.
- If X is greater than this value, then search the middle of the right half next.
- Continue in this manner.

Pseudocode:

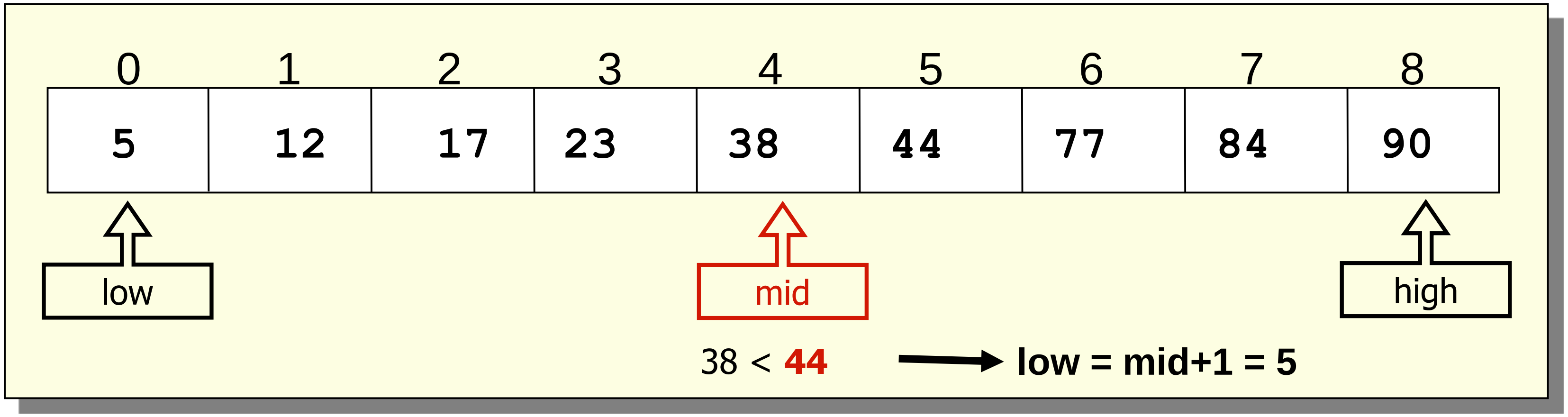
```
If ( value == middle element )
     value is found
else if ( value < middle element )
    search left-half of list with the same method
else
    search right-half of list with the same method
```

Sequence of Successful Search - 1

search(44)

	low	high	mid
#1	0	8	4

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



0	1	2	3	4	5	6	7	8
5	12	17	23	38	44	77	84	90

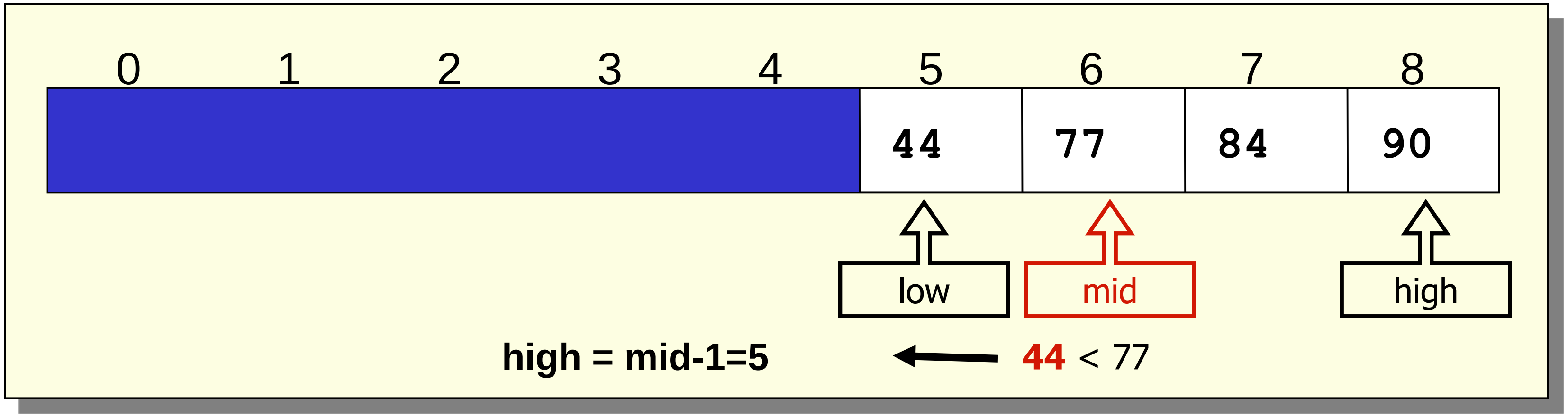
38 < **44** → **low = mid+1 = 5**

Sequence of Successful Search - 2

	low	high	mid
#1	0	8	4
#2	5	8	6

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



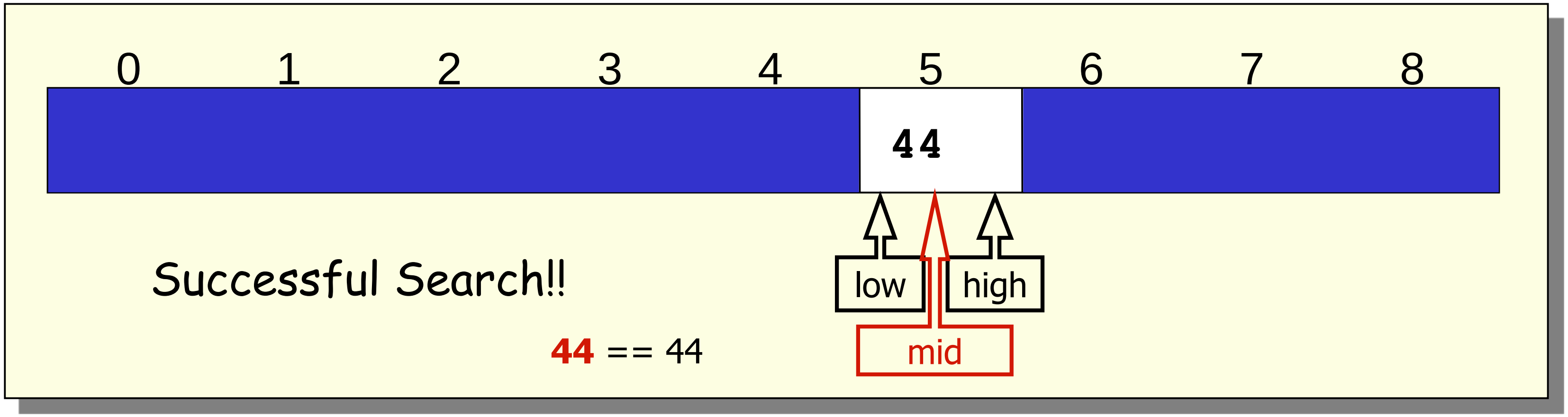
high = mid-1=5 ← 44 < 77

Sequence of Successful Search - 3

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



Successful Search!!

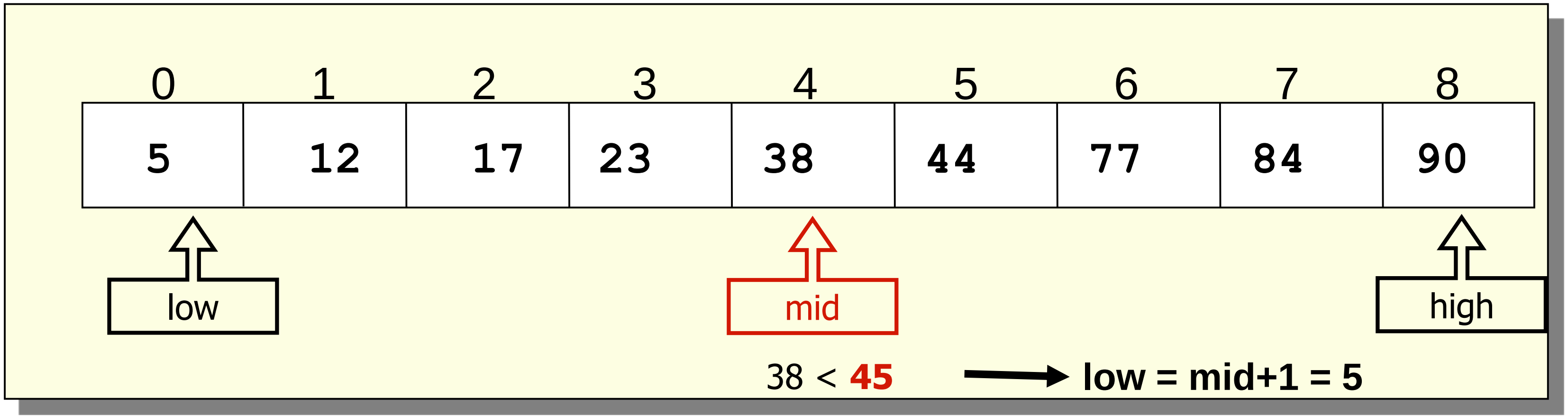
44 == 44

Sequence of Unsuccessful Search - 1

	low	high	mid
#1	0	8	4

search(45)

$$mid = \left\lfloor \frac{low+high}{2} \right\rfloor$$



0	1	2	3	4	5	6	7	8
5	12	17	23	38	44	77	84	90

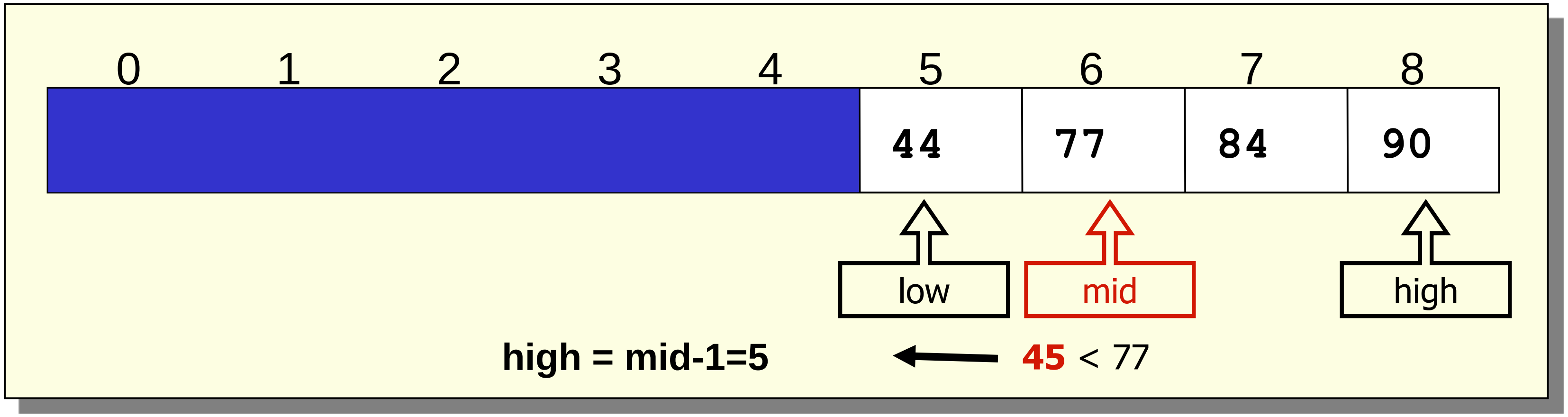
38 < **45** → **low = mid+1 = 5**

Sequence of Unsuccessful Search - 2

	low	high	mid
#1	0	8	4
#2	5	8	6

search(45)

$$mid = \left\lfloor \frac{low+high}{2} \right\rfloor$$



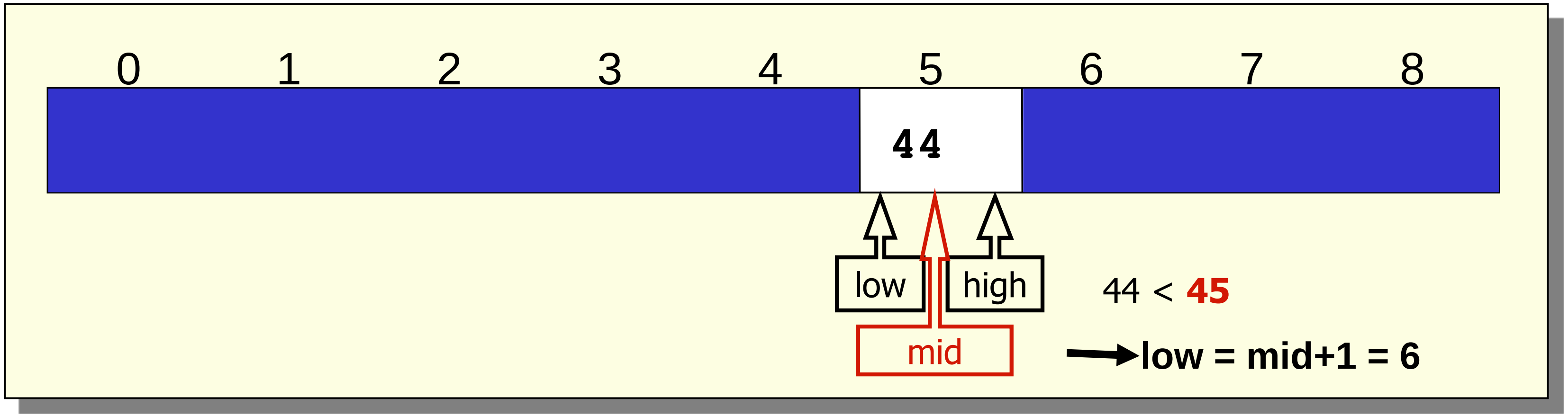
high = mid-1=5 ← **45** < 77

Sequence of Unsuccessful Search - 3

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

search(45)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



44 < **45**

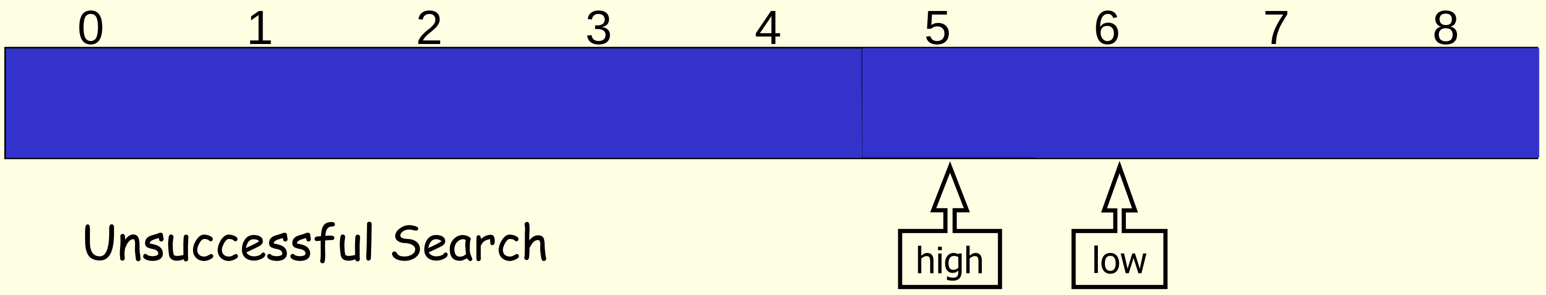
→ **low = mid+1 = 6**

Sequence of Unsuccessful Search - 4

search(45)

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5
#4	6	5	

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



Unsuccessful Search

no more elements to search ← low > high

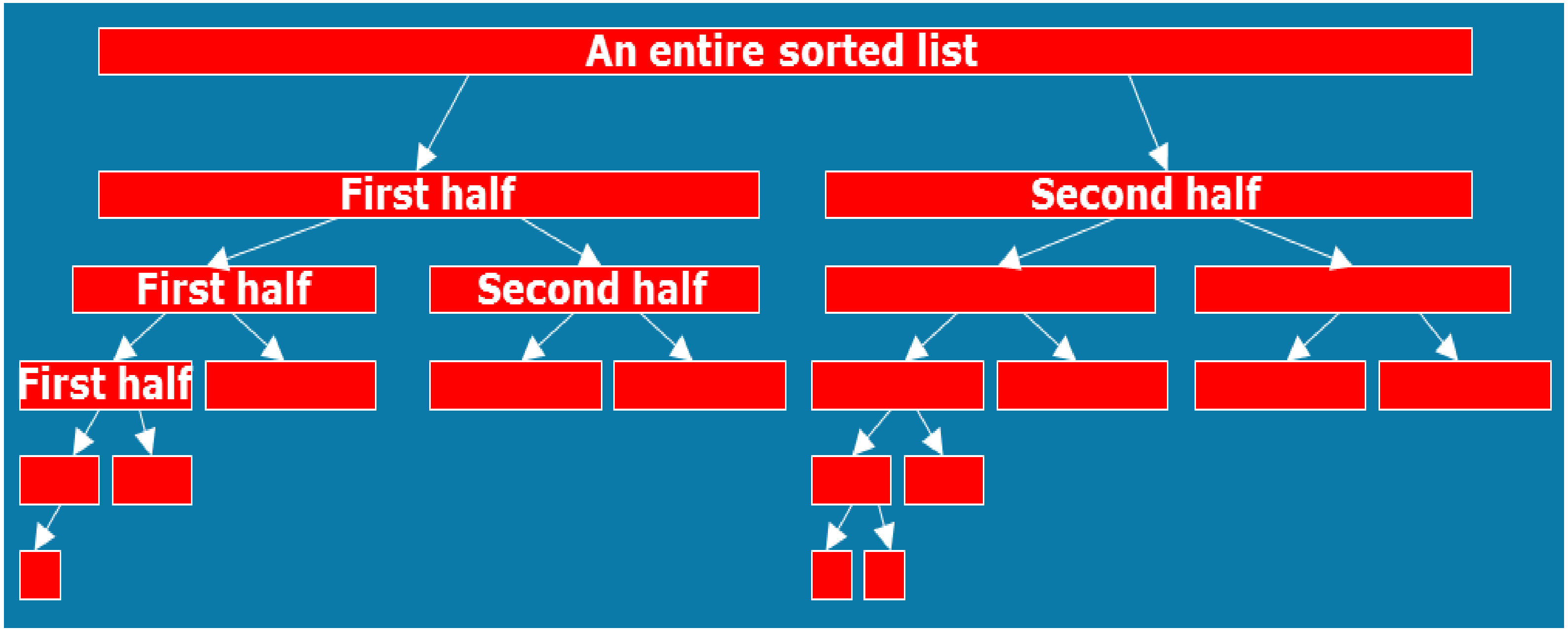
Implementation:

```
void binarySearch(int arr[], int val, int N)
{
   int low = 0;
   int high = N - 1;
   int mid;
   while ( low <= high ){
        mid = ( low + high )/2;
        if (arr[mid]== val)
           cout<<“found at position”<<mid; //found!

        else if (arr[mid] < val)
           low = mid + 1;
        else
           high = mid - 1;
        }
    cout<<“Element not found”<<endl;
}
```

Binary tree:

- The search divides a list into two small sub-lists till a sub-list is no more divisible.



Efficiency:

- After 1 bisection $N/2$ items
- After 2 bisections $N/4 = N/2^2$ items
- . . .
- After *i* bisections $N/2^i = 1$ item

$O(\log_2 N)$