Exponential Search

# Exponential Search

An exponential search (also called doubling search or galloping search or Struzik search) is an algorithm, created by Jon Bentley and Andrew Chi-Chih Yao in 1976, **for searching sorted, unbounded/infinite lists**.

There are numerous ways to implement this with the most common being to determine a range that the search key resides in and performing a binary search within that range.

Exponential search can also be used to search in bounded lists. Exponential search can even out-perform more traditional searches for bounded lists, such as binary search, when the element being searched for is near the beginning of the array. This is because exponential search will run in O(log i) time, where i is the index of the element being searched for in the list, whereas binary search would run in O(log n) time, where n is the number of elements in the list.

**By looking at the time complexity O(log i), it should not be confused with the binary search.**

# Basic Algorithm

The algorithm consists of two stages:

1. Find range where element is present
2. Do Binary Search in above found range

In the first stage, assuming that the list is sorted in ascending order, the algorithm looks for the first exponent, j, where the value 2j is greater than the search key. This value, 2j becomes the upper bound for the binary search with the previous power of 2, 2j - 1, being the lower bound for the binary search.

**How to find the range where element may be present?**

The idea is to start with subarray size 1 compare its last element with x, then try size 2, then 4 and so on until last element of a subarray is not greater.  
Once we find an index i (after repeated doubling of i), we know that the element must be present between i/2 and i (Why i/2? because we could not find a greater value in previous iteration)

In each step, the algorithm compares the search key value with the key value at the current search index. If the element at the current index is smaller than the search key, the algorithm repeats, skipping to the next search index by doubling it, calculating the next power of 2. If the element at the current index is larger than the search key, the algorithm now knows that the search key, if it is contained in the list at all, is located in the interval formed by the previous search index, 2j - 1, and the current search index, 2j. The binary search is then performed with the result of either a failure, if the search key is not in the list, or the position of the search key in the list.

# Pseudocode

// Returns the position of key in the array arr of length size

template <typename T>

int exponential\_search(T arr[], int size, T key)

{

if (size == 0) {

return NOT\_FOUND;

}

int bound = 1;

while (bound < size && arr[bound] < key) {

bound \*= 2;

}

return binary\_search(arr, key, bound/2, min(bound, size));

}

# C/C++ Implementation

// Returns position of first ocurrence of x in array

int exponentialSearch(int arr[], int n, int x)

{

    // If x is present at firt location itself

    if (arr[0] == x)

        return 0;

    // Find range for binary search by repeated doubling

    int i = 1;

    while (i < n && arr[i] <= x)

        i = i\*2;

    //  Call binary search for the found range.

    return binarySearch(arr, i/2, min(i, n), x);

}

# Complexity

**Best case**:

**Average case**:

**Worst case**:

\* **Auxiliary Space:**

* recursive and requires O(log n) space
* iterative Binary Search, need only O(1) space

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Best case** | **Average case** | **Worst case** | **Space Complexity** |
| Linear Search | O(1) | O(n) | O(n) | O(1) |
| Binary Search | O(1) | O(logn) | O(logn) | O(1)\* |
| Jump Search | O(1) | O(√ n) | O(√ n) | O(1) |
| Interpolation Search | O(1) | O (log log n)) | O(n) | O(1) |
| Exponential Search | O(1) | O(log i) | O(log i) | O(1) |
| Fibonacci Search | O(1) | O(logn) | O(logn) | O(1) |

# Application

1. Exponential Binary Search is particularly useful for unbounded searches, where size of array is infinite. Please refer Unbounded Binary Search for an example.
2. It works better than Binary Search for bounded arrays also when the element to be searched is closer to the first element.

# Example

## Recursive Search

// C++ program to find an element x in a sorted array using Exponential search.

#include <bits/stdc++.h>

using namespace std;

int binarySearch(int arr[], int, int, int);

// Returns position of first ocurrence of x in array

int exponentialSearch(int arr[], int n, int x)

{

// If x is present at firt location itself

if (arr[0] == x)

return 0;

// Find range for binary search by repeated doubling

int i = 1;

while (i < n && arr[i] <= x)

i = i\*2;

// Call binary search for the found range.

return binarySearch(arr, i/2, min(i, n), x);

}

// A recursive binary search function. It returns location of x in given array arr[l..r] is present, otherwise -1

int binarySearch(int arr[], int l, int r, int x)

{

if (r >= l)

{

int mid = l + (r - l)/2;

// If the element is present at the middle itself

if (arr[mid] == x)

return mid;

// If element is smaller than mid, then it can only be present n left subarray

if (arr[mid] > x)

return binarySearch(arr, l, mid-1, x);

// Else the element can only be present in right subarray

return binarySearch(arr, mid+1, r, x);

}

// We reach here when element is not present in array

return -1;

}

// Driver code

int main(void)

{

int arr[] = {2, 3, 4, 10, 40};

int n = sizeof(arr)/ sizeof(arr[0]);

int x = 10;

int result = exponentialSearch(arr, n, x);

(result == -1)? printf("Element is not present in array")

: printf("Element is present at index %d",

result);

return 0;

}

Output :

Element is present at index 3