Fundamental Algorithms and Data Structures for H2 Computing Practical

# Searching Algorithms

## Linear Search

### Description

A linear search algorithm checks all elements of an array **one by one**, and **in sequence**, until the **desired result is found**. It can be used for both sorted and unsorted arrays.

**Time complexity:** ***O(N)***

### Code in Python

def LinearSearch(AR, SearchKey)

flag = False

for i in range(len(AR)):

if AR[i] == SearchKey:

flag = True

break

return flag

## Binary Search

### Description

Binary search is an algorithm that works on the principle of **divide and conquer**, that involves **iteration** or **recursion**. The array is **split at the middle of the array**, creating two sub-arrays. Depending on the condition, either the **left sub-array** or the **right sub-array** is chosen, essentially cutting the size of the array by half. This process is repeated until the element is found. Binary search can only be used for arrays sorted in a particular order (e.g. numerical/alphabetical order).

**Time complexity:** ***O(log N)***, making it more efficient than a linear search.

### Code in Python

def BinarySearch(AR, SearchKey):

found = False

low = 0

high = len(AR) - 1

while ((not found) and (low <= high)):

middle = int((low + high)/2)

if AR[middle] = SearchKey:

ElementFound = True

else:

if SearchKey < AR[middle] THEN

high = middle - 1

else:

low = middle + 1

return ElementFound

# Sorting Algorithms

## Bubble Sort

### Description

**Bubble sort** is a simple sorting algorithm that **compares** **adjacent elements** and **swaps them** depending on whether the elements are **out of order** in each pass.

After *N* passes, the last *N* elements are in the **correct position**.

Therefore, *N* - 1 passes are needed to sort *N* elements in their **correct positions**.

**Time complexity:** ***O(N2)***

### Code in Python

def BubbleSort(AR):

swapped = True

while swapped == True:

swapped = False

for i in range(size(AR)):

if AR[i] > AR[i + 1]:

temp = AR[i]

AR[i] = AR[i + 1]

AR[i + 1] = temp

swapped = True

else:

i += 1 # increment i

### Demonstration

|  |  |  |
| --- | --- | --- |
| Array to sort: | **24, 97, 57, 77, 6, 41, 90** |  |
| After 1st pass: | **24, 57, 77, 6, 41, 90, 97** |  |
| After 2nd pass: | **24, 57, 6, 41, 77, 90, 97** |  |
| After 3rd pass: | **24, 6, 41, 57, 77, 90, 97** |  |
| After 4th pass: | **6, 24, 41, 57, 77, 90, 97** |  |
| After 5th pass: | **6, 24, 41, 57, 77, 90, 97** | **(no swaps occurs)** |
| After 6th pass: | **6, 24, 41, 57, 77, 90, 97** | **(no swaps occurs)** |
| After 7th pass: | **6, 24, 41, 57, 77, 90, 97** | **(no swaps occurs)** |

## Insertion Sort

### Description

**Insertion sort** is a sorting algorithm that **partitions the array into two parts**: a **sorted sub-array** and an **unsorted sub-array**. Initially, the sorted sub-array consists of the first element, and the unsorted sub-array consists of the rest of the elements.

During each iteration, the first element of the unsorted sub-array is compared with the elements of the sorted sub-array, and inserted into the sorted sub-array. This increases the size of the sorted sub-array by 1, and decreases the size of the unsorted sub-array by 1.

**Time complexity:** **O(N2)**

### Code in Python

def InsertionSort(AR):

for i in range(1, len(AR)):

j = i - 1

temp = AR[i]

while (j >= 0) and (temp < AR[j]):

AR[j + 1] = AR[i]

j -= 1 # decrement i

AR[j + 1] = temp

**Side note:** *Although bubble sort and insertion sort both have a time complexity of O(N2), insertion sort is typically twice as fast as bubble sort.*

### Demonstration

**Green** represents the sorted sub-array; **red** represents the unsorted sub-array.

|  |  |  |
| --- | --- | --- |
| Array to sort: | **24, 97, 57, 77, 6, 41, 90** |  |
| After 1st pass: | **24, 97, 57, 77, 6, 41, 90** | (no swaps occur) |
| After 2nd pass: | **24, 57, 97, 77, 6, 41, 90** |  |
| After 3rd pass: | **24, 57, 77, 97, 6, 41, 90** |  |
| After 4th pass: | **6, 24, 57, 77, 97, 41, 90** |  |
| After 5th pass: | **6, 24, 41, 57, 77, 97, 90** |  |
| After 6th pass: | **6, 24, 41, 57, 77, 90, 97** |  |

## Quicksort

### Description

**Quicksort** is a sorting algorithm that uses the principle of **divide and conquer** to arrange elements of an array into their **correct positions**, using a **pivot** that divides the array into two sub-arrays.

**Time complexity:** **O(N log N)**

1. The algorithm goes through the left sub-array and finds any element that **belongs** in the right sub-array by comparing with the **pivot**.
2. Then, the algorithm goes through the right sub-array and finds any element that **belongs** in the left sub-array by comparing with the **pivot**.
3. The algorithm then swaps the value of the elements belonging to the **wrong sub-array**.
4. As a result, after one pass, all the elements of the left sub-array are **less than** the value of the **pivot**, and all the elements of the right sub-array are **greater than** the value of the **pivot**. (depends on implementation)
5. This whole process **carries on** within the left sub-array, then within the right sub-array **recursively** (from steps 1 to 5).
6. In the end, a **sorted array** is obtained.

### Pivot

The **pivot** can be any element of the array, although the **best** element to choose as the pivot is usually the **middle element**, with its index calculated as

.

### Code in Python

def Partition(ar, left, right):

i = left + 1

j = right

focus = left

while i <= j:

while ar[i] < ar[focus]:

i += 1

while ar[j] > ar[focus]:

j -= 1

if i <= j:

ar[i], ar[j] = ar[j], ar[i]

i += 1

j -= 1

else:

ar[j], ar[focus] = ar[focus], ar[j]

return j

def QuickSort(ar, left, right):

if left < right:

focus = Partition(ar, left, right)

QuickSort(ar, left, focus - 1)

QuickSort(ar, focus + 1, right)

return ar

To run the quicksort: QuickSort(array, 0, len(array) - 1)

# Data Structures

## Linked List

A linked list is a data structure that