SEQUENTIAL SEARCH

// Searches for a given value in a given array by sequential search
// Input: An array A[0..n-1] & a search key K
// output: The index of the 1st element of A that matches K or -1 if there are no matching elements

```
i ← 0
while i<n and A[i]!=K do
      i ← i+1
if i<n return i
   else return -1
```

W.C → n key comparisions
B.C → 1 comparision
A.C → (n+1)/2, K is in A
Space.C: O(1)

BINARY SEARCH:

// Input: Sorted array a[i] < .... < a[j] and key x;

```
m ← (i+j)/2
while i<j and x != a[m]
      if x<a[m] then j ← m-1
                else i ← m+1
  if x = a[m] then output a[m];
```

O(logn)

LINEAR SEARCH:

```
A[10], K, i=0;
while (i<n) do
      if (a[i]==k)
          then return i;
      else i++;
if (i>=n)  print("Element not found");
```

MAXIMUM ELEMENT:

```
maxval ← A[0]
for i ← 1 to i ← n-1 do
   if A[i] > maxval then maxval ← A[i]
return maxval
```

ELEMENT UNIQUENESS PROBLEM:

```
for i ← 0 to i ← n-2 do
    for j ← i-1 to j ← n-1 do
        if A[i] = A[j] return False
return true
```

$T(n) = \sum_{0 \le i \le n-2} \left(\sum_{i+1 \le j \le n-1} 1 \right)$

$= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i-1)+1 = \sum_{i=0}^{n-2} (n-i-1)$

$= (n-1)+(n-2)+(n-3)+\cdots+(n-n-2-1)$

$= (n-1)+(n-2)+\cdots+(1)$

$= \frac{(n-1)(n)}{2} \Rightarrow O(n^2)$

Space complexity: $O(1)$

Time complexity: $O(n^2)$

TOWER OF HANOI:

```
Procedure Hanoi(disk, source, dest, aux)
  IF disk == 1, then
      move disk from source to dest
  ELSE
      Hanoi(disk-1, source, aux, dest)
      move disk from source to dest
      Hanoi(disk-1, aux, dest, source)
  END IF
END Procedure
```

Recurrence for no. of moves $\Rightarrow 2^n - 1$

$M(n) = 2M(n-1)+1$

SESSION 4

CHAPTER 3
BRUTE FORCE

Brute force – problem statement, definitions of concepts

SELECTION SORT:

$A_0 \le A_1 \le \cdots \le A_{i-1} \mid A_i, \ldots, A_{min}, \ldots A_{n-1}$

$A_0 \le A_1 \le \cdots \le A_{i-1}$: in their final positions; $A_i, \ldots, A_{min}, \ldots A_{n-1}$: the last $n-i$ elements

After $n-1$ passes, the list is sorted

```
for i ← 0 to i ← n-2 do
    min ← i
    for j ← i+1 to j ← n-1 do
        if A[j] < A[min] min ← j
    swap A[i] and A[min]
```

$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{(n-1)n}{2} = \Theta(n^2)$

space complexity: $O(1)$ inplace sorting

BUBBLE SORT:

```
for i ← 0 to n-2 do
    for j ← 0 to n-2-i do
        if A[j+1] < A[j] swap A[j] and A[j+1]
```

$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$ $\quad \Theta(n^2)$ W.C

B.C $O(n)$ already sorted

Space Com.: $O(1)$ inplace sorting

SEQUENTIAL SEARCH (above)

↳ LINEAR SEARCH

B.C $O(1)$

A.C $O(n/2)$

W.C $O(n)$

STRING MATCH:

```
B.FStringMatch (T[0...n-1], P[0...m-1])
for i ← 0 to n-m do
    j ← 0
  while j<m and P[j] = T[i+j] do
        j ← j+1
    if j=m return i
return -1
```

EXHAUSTIVE SEARCH

— TRAVELING SALESMAN PROBLEM:

— KNAPSACK PROBLEM

- ASSIGNMENT PROBLEM

```
DFS(G)
count ← 0
for each vertex v in V do
    if v is marked with 0
      dfs(v)
dfs(v)
count ← count + 1 ; mark v with count
for each vertex w in V adj to v do
    if w is marked with 0
      dfs(w)
```

Time complexity : $O(V+E)$

BFS: BFS(G)

```
count ← 0
for each vertex v in V do
  if v is marked with 0
      bfs(v)
bfs(v)
count ← count + 1 ; mark v with count & initialize a queue with v
while the queue is not empty do
    for each vertex w in V adj to the front vertex do
      if w is marked with 0
          count ← count+1 ; mark w with count
          add w to the queue
    remove the front vertex from the queue
```

Time complexity : $O(V+E)$

CHAPTER 4
DECREASE & CONQUER

1) decrease by const (1)

2) decrease by a const factor (half)

3) Variable size decreases (varies iteration by iteration)

INSERTION SORT:

```
for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v
```

$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} [i-1+1] = \sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} \in \Theta(n^2)$

$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$

$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$

BINARY SEARCH (above):

CHAPTER 5
DIVIDE & CONQUER

$T(n) = aT(\frac{n}{b}) + \Theta(n^k \log^p n)$ $a \geq 1, b > 1, k \geq 0$, p is a real no.

Case 01:

If $a > b^k$

$T(n) = \Theta(n^{\log_b a})$

Case 02:

If $a = b^k$ and

if $p < -1$, $T(n) = \Theta(n^{\log_b a})$

if $p = -1$, $T(n) = \Theta(n^{\log_b a} \cdot \log^2 n)$

if $p > -1$, $T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$

case 03:

If $a < b^k$ and

if $p < 0$, $T(n) = \Theta(n^k)$

if $p \geq 0$, $T(n) = \Theta(n^k \log^p n)$

MERGESORT:

- Mergesort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array A[0..n − 1] by dividing it into two halves A[0..n/2 − 1] and A[n/2..n − 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM *Mergesort*(A[0..n − 1])

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)    //see below
```

ALGORITHM *Merge*(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

PRESORTING

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$
//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for $i \leftarrow 0$ **to** $n-2$ **do**
  if $A[i] = A[i+1]$ **return false**
return true

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**
return true

COMPUTING A MODE:

ALGORITHM *PresortMode*$(A[0..n-1])$
//Computes the mode of an array by sorting it first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: The array's mode
sort the array A
$i \leftarrow 0$ //current run begins at position i
$modefrequency \leftarrow 0$ //highest frequency seen so far
while $i \leq n-1$ **do**
  $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
  while $i + runlength \leq n-1$ **and** $A[i+runlength] = runvalue$
    $runlength \leftarrow runlength + 1$
  if $runlength > modefrequency$
    $modefrequency \leftarrow runlength$; $modevalue \leftarrow runvalue$
  $i \leftarrow i + runlength$
return $modevalue$

Ex: 5, 1, 5, 7, 6, 5, 7

1555677

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

4

ALGORITHM *HeapBottomUp*$(H[1..n])$
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array $H[1..n]$ of orderable items
//Output: A heap $H[1..n]$
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
  $k \leftarrow i$; $v \leftarrow H[k]$
  $heap \leftarrow$ **false**
  while not $heap$ **and** $2 * k \leq n$ **do**
    $j \leftarrow 2 * k$
    if $j < n$ //there are two children
      if $H[j] < H[j+1]$ $j \leftarrow j+1$
    if $v \geq H[j]$
      $heap \leftarrow$ **true**
    else $H[k] \leftarrow H[j]$; $k \leftarrow j$
  $H[k] \leftarrow v$