Algorithms – Practical Content Review C

- **Linear Search**
  - Problem:
    - Given an array of objects, A, and a target object, t, return the index of any instance of t in A if t exists in A, else will return -1.
  - Complexity
    - Worst Case: target not in list → O(n)
  - Algorithm

```
FUNCTION LINEARSEARCH(A: ARRAY OF INTEGER, t: INTEGER) RETURNS INTEGER
        DECLARE index: INTEGER
        index ← -1
        FOR i = 1 TO A.SIZE
                IF A[i] = t THEN
                        index ← i
                        BREAK
                ENDIF
        ENDFOR
        RETURN index
ENDFUNCTION
```

  - Variations:
    - Search target requires a different criteria (not just object existence).
    - Must find all instances of target.
    - Must find particular instance of target (first, last, etc.).
    - Must find object just greater/smaller than target.

- **Binary Search**
  - Problem: same as linear search
  - Complexity
    - Worst Case: target not in list → O(log n)
  - Algorithm:

```
FUNCTION BINARYSEARCH(A: ARRAY OF INTEGER, t: INTEGER) RETURNS INTEGER
        DECLARE start, mid, end: INTEGER
        start ← 1
        end ← A.SIZE
        WHILE start <= end DO
                mid ← (start + end) DIV 2
                IF t = A[mid] THEN
                        RETURN mid
                ENDIF
                IF t < A[mid] THEN
                        end ← mid - 1
                ELSE
                        start ← mid + 1
                ENDIF
        ENDWHILE
        RETURN -1
ENDFUNCTION
```

  - Variations: same as linear search.

- **Base Conversion (denary to base k)**
    - Problem 1:
        - Given a positive integer value, d, and another positive integer value k, where k typically in range [2, 16] (or at most in the range [2, 62]), return a string corresponding to the representation of d as a base k number.
    - Complexity:
        - Worst Case: No variation with cases. O(log n).
    - Algorithm:

```
FUNCTION D2K(d: INTEGER, k: INTEGER) RETURNS STRING
       DECLARE result, mapping: STRING
       mapping ← "0123456789ABCDEF"
       result ← ""
       WHILE d > 0 DO
              result ← CONCATENATE(mapping[(d MOD k) + 1], result)
              d ← d DIV k
       ENDWHILE
       RETURN result
ENDFUNCTION
```

    - Variations: None. Typically, extensions are above its application.

- **Base Conversion (base k to denary)**
    - Problem 2:
        - Given a string representing a positive integer value in base k, s, and another positive integer value k, where k typically in range [2, 16] (or at most in the range [2, 62]), return an integer representing the denary value of s.
    - Complexity:
        - Worst Case: No variation with cases. O(|s|).
    - Algorithm:

```
FUNCTION D2K(s: STRING, k: INTEGER) RETURNS INTEGER
       DECLARE result, temp: INTEGER
       DECLARE mapping: STRING
       mapping ← "0123456789ABCDEF"
       result ← 0
       FOR i = 1 TO s.SIZE
              temp ← (mapping.GETINDEX(s[i]) – 1) * k ^ (s.SIZE – i)
              result ← result + temp
       ENDFOR
       RETURN result
ENDFUNCTION
```

    - Variations: None. Typically, extensions are above its application.

- **Bubble Sort**
  - Problem:
    - Given an array of objects, A, re-arrange the objects in A such that they are sorted – i.e., such that for any A[i], A[j] in A, if i < j, then A[i] ≤ A[j].
  - Complexity:
    - Worst Case: Sorted in reverse order to requirement. $O(|A|^2)$.
  - Algorithm:

```
FUNCTION BUBBLESORT(A: ARRAY OF INTEGER) RETURNS ARRAY OF INTEGER
    DECLARE swap: BOOLEAN
    DECLARE temp: INTEGER
    FOR i = 1 to A.SIZE – 1
        swap ← FALSE
        FOR j = 1 to A.SIZE – i
            IF A[j] > A[j + 1] THEN
                temp ← A[j]
                A[j] ← A[j + 1]
                A[j + 1] ← temp
                swap ← TRUE
            ENDIF
        ENDFOR
        IF NOT swap THEN
            BREAK
        ENDIF
    ENDFOR
    RETURN A
ENDFUNCTION
```

  - Variations:
    - Descending instead of the typical ascending order.
    - More complex expression for object (i.e., elements in A) comparison.
    - Applications of sorting – e.g., calculation of median, quartiles, etc.

- **Insertion Sort**
  - Problem:
    - Same as Bubble Sort.
  - Complexity:
    - Worst Case: Sorted in reverse order to requirement. $O(|A|^2)$.
  - Algorithm:

```
FUNCTION INSERTIONSORT(A: ARRAY OF INTEGER) RETURNS ARRAY OF INTEGER
    DECLARE j, temp: INTEGER
    FOR i = 2 to A.SIZE
        j ← i
        WHILE j > 1 AND A[j] < A[j – 1] DO
            temp ← A[j]
            A[j] ← A[j - 1]
            A[j - 1] ← temp
            j ← j - 1
        ENDWHILE
    ENDFOR
    RETURN A
ENDFUNCTION
```

  - Variations:
    - Same as Bubble Sort.

- **Quicksort**
  - Problem:
    - Same as Bubble Sort.
  - Complexity:
    - Worst Case: Pivot selection always selects largest or smallest element. $O(|A|^2)$
  - Algorithm:

```
FUNCTION QUICKSORT(A: ARRAY OF INTEGER) RETURNS ARRAY OF INTEGER
       IF A.SIZE < 2 THEN
              RETURN A
       ENDIF

       DECLARE pivot: INTEGER
       DECLARE less, more: LINKEDLIST OF INTEGER
       pivot ← A[1]
       FOR i = 2 to A.SIZE
              IF A[i] < pivot THEN
                     less.INSERT(A[i])
              ELSE
                     more.INSERT(A[i])
              ENDIF
       ENDFOR
       RETURN CONCATENATEARRAY(QUICKSORT(ARRAY(less)), ARRAY(pivot),
                               QUICKSORT(ARRAY(more)))

ENDFUNCTION
```

  - Variations:
    - Same as Bubble Sort.
    - The above implementation of the quicksort algorithm does not sore “in place”, and has a high space complexity. In order to overcome this, you need to change the algorithm slightly – i.e., use a variant that does not create new linked lists to store elements greater/less than the pivot.