ALGORITHM BinarySearch (A[0..n-1], n, x)

INPUT: Array A of n floating-point numbers, and target x

OUTPUT: Index i such that A[i] = x, or -1 if x not found

```
Iterative:
    low <- index of first element
    high <- index of last element                    # define low and high bounds for search space

    do:
        mid <- index of the "middle" element         # calculate the bisect (middle) of the search space
        midVal <- A[mid]
        if  (midVal == x):                           # found x, return now
        return mid;
        else:                                        # didn't find x, divide and conquer from here
        if (midVal < x):            # remove the low-end (search space divided in half)
            low <- mid + 1          # search to the right of mid
        else:                       # remove the high-end (search space divided in half)
            high <- mid - 1                          # Search to the left of mid
    while:
        low <= high                                  # run until low and high converge (no x is found)

    return -1                                        # no index was found
```

Recursive:

```
recursiveSearch(low <- index of first element, high <- index of last element):

if (low >= high):                          `           # run until low and high converge (no x is found)
    return -1                                          # base case #1, no such element found

mid <- index of the middle element                     # calculate the bisect (middle) of the search space
midVal <- A[mid]

if (midVal == x):                                      # base case #2, x found at mid
    return mid
else:                                                  # didn't find x, divide and conquer from here
    if (midVal < x):                               # remove the low-end (search space divided in half)
        recursiveSearch(mid + 1, high)  # Search to the right of mid
    else:                                              # remove the high-end (search space divided in half)
        recursiveSearch(low, mid - 1)       # Search to the left of mid
```

```
ALGORITHM DoSomething(n)

INPUT: A positive integer n

OUTPUT: s, the summation of each element cubed in the set ℤ {0…n}


IF n == 1                                            // base case
    return 1                                         // starts with 1
ELSE
    return DoSomething(n-1) + n * n * n// 1 + 2^3
                                                     // 9 + 3^3
                                                     // 36 + 4^3
                                                     // 100 + 5^3
                                                     // 225 + 6^3
                                                     // 441 + 7^3…
```

Part 2.1:

This algorithm finds the summation of cubes for numbers n to 1;

The algorithm outputs s, the summation of each element cubed in the set ℤ {0…n}

Part 2.2:

The Recursive Case:

T() = T(n-1) + 1, T(1) = 1 // T(n-1) -> find each

Part 2.3:

```
// ==============
// Recursive Case
// ==============
T(n) = T(n-1) + 1


// =============
// 1st Occurence
// =============
T(n-1)  = T((n - 1) - 1) + 1 + 1
            = T(n - 2) + 2


// =============
// 2nd Occurence
// =============
T(n-2)  = T((n - 2) - 1) + 2 + 1
            = T(n - 3) + 3


// =============
// ith Occurence
// =============
T(n) = T(n - i) + i
            // substitute i = n - 1 for the
            = T(n - (n - 1)) + n - 1
            = T(n - n + 1) + n - 1
```

```
          // ===========

          // Closed Form

          // ===========

          = T(1) + n - 1


// ===============

// Time Complexity

// ===============

T(n) = O(n)
```

Part 2.4:

The space complexity of the DoSomething algorithm is O(n) because each recursion reduces n by 1 until the base case when n is 1, and each recursion only adds a single function to the heap.

Part 2.5:

```
ALGORITHM DoSomething(n)

INPUT: A positive integer n

OUTPUT: s, the summation of each element cubed in the set ℤ{0…n}


sum -> holds sum of all cubed n

IF n > 1 DO:

    sum += n * n * n;

    n--;

return sum
```

Part 3.1:

1.

```
// ===================
// Recurrence Relation
// ===================
T(n) = T(n − 1) + n, T(1) = 1

// ==============
// Recursive Case
// ==============
T(n) = T(n − 1) + n

// =============
// 1st Occurrence
// =============
T(n) = T((n − 1) − 1) + (n − 1) + n
  = T(n − 2) + (n − 1) + n

// =============
// 2nd Occurrence
// =============
T(n) = T((n − 2) − 1) + (n − 2) + (n − 1) + n
  = T(n − 3) + (n − 2) + (n − 1) + n

// =============
// ith Occurrence
// =============

T(n) = T(n − i) + sum(n − k)[from k = 0 to i − 1]


        // ===========

        // Closed Form

        // ===========

        // substitute i = n − 1 ⇒ n − i = 1

        T(n) = T(1) + (1 + 2 + 3 + ... + n)

             = 1 + n(n + 1)/2


// ===============

// Time Complexity

// ===============

T(n) = O(n^2)
```

2.

```
// ===================
// Recurrence Relation
// ===================
T(n) = 2T(n/2) + n, T(1) = 1


// ==============
// Recursive Case
// ==============
T(n) = 2T(n/2) + n


// =============
// 1st Occurrence
// =============
T(n) = 2(2T(n/2^2) + n/2) + n
   = 2^2 T(n/2^2) + 2n


// =============
// 2nd Occurrence
// =============
T(n) = 2^3 T(n/2^3) + 3n


// =============
// ith Occurrence
// =============
T(n) = 2^i T(n/2^i) + i·n
```

```
    // ===========

    // Closed Form

    // ===========

    // substitute n/2^i = 1 ⇒ i = log_2(n)

    T(n) = 2^log_2(n)·T(1) + n log_2(n)

         = n + n log n


// ===============

// Time Complexity

// ===============

T(n) = O(n log n)
```

3.

```
// ===================
// Recurrence Relation
// ===================
T(n) = 4T(n/2) + n^2, T(1) = 1

// ==============
// Recursive Case
// ==============
T(n) = 4T(n/2) + n^2

// =============
// 1st Occurrence
// =============
T(n) = 4(4T(n/2^2) + (n/2)^2) + n^2
   = 4^2 T(n/2^2) + 2n^2

// =============
// 2nd Occurrence
// =============
T(n) = 4^3 T(n/2^3) + 3n^2

// =============
// ith Occurrence
// =============
T(n) = 4^i T(n/2^i) + sum(n^2)[from j = 0 to i − 1]
```

```
    // ===========
    // Closed Form
    // ===========
    // substitute n/2^i = 1 ⇒ i = log_2(n)
    T(n) = 4^log_2(n)·T(1) + n^2 log_2(n)
         = n^2 + n^2 log n


// ===============
// Time Complexity
// ===============
T(n) = O(n^2 log n)
```

4.

```
// ====================
// Recurrence Relation
// ====================
T(n) = 8T(n/2) + 1, T(1) = 1


// ==============
// Recursive Case
// ==============
T(n) = 8T(n/2) + 1


// =============
// 1st Occurrence
// =============
T(n) = 8(8T(n/2^2) + 1) + 1
   = 8^2 T(n/2^2) + 9


// =============
// 2nd Occurrence
// =============
T(n) = 8^3 T(n/2^3) + (1 + 8 + 8^2)


// =============
// ith Occurrence
// =============
T(n) = 8^i T(n/2^i) + sum(8^j)[from j = 0 to i − 1]
```

```
        // ===========
        // Closed Form
        // ===========
        // substitute n/2^i = 1 ⇒ i = log_2(n)
        T(n) = 8^log_2(n)·T(1) + O(8^log_2(n))
             = n^3 + O(n^3)

// ===============
// Time Complexity
// ===============
T(n) = O(n^3)
```

5.

```
// ===================
// Recurrence Relation
// ===================
T(n) = 2T(n/2) + sqrt(n), T(1) = 1

// ==============
// Recursive Case
// ==============
T(n) = 2T(n/2) + sqrt(n)

// =============
// 1st Occurrence
// =============
T(n) = 2(2T(n/2^2) + sqrt(n/2)) + sqrt(n)
   = 2^2 T(n/2^2) + 2 sqrt(n/2) + sqrt(n)

// =============
// 2nd Occurrence
// =============
T(n) = 2^3 T(n/2^3) + 2^2 sqrt(n/2^2) + 2 sqrt(n/2) + sqrt(n)

// =============
// ith Occurrence
// =============
T(n) = 2^i T(n/2^i) + sum(2^j sqrt(n/2^j))[from j = 0 to i − 1]
```

```
    // ===========
    // Closed Form
    // ===========
    // substitute n/2^i = 1 ⇒ i = log_2(n)
    2^i T(1) = 2^log_2(n) = n


    2^j sqrt(n/2^j) = sqrt(n)·2^{j/2}


    sum(2^{j/2}) = O(2^{(log n)/2}) = O(sqrt(n))


    T(n) = n + sqrt(n)·sqrt(n)
         = n + n
         = 2n


// ===============
// Time Complexity
// ===============
T(n) = O(n)
```

Part 3.2:

1.

```
// ===================
// Recurrence Relation
// ===================
T(n) = 2T(n/2) + n


// ==============
// Master Theorem Case
// ==============
// T(n) = aT(n/b) + c*n^k
// a = 2, b = 2, c = 1, k = 1
// T(1) = 1
// b^k = 2^1
// a = 2
// Case 2: a = b^k; T(n) = O(n^k * log(n))
// T(n) = O(n^1 * log(n))


// ===============
// Time Complexity
// ===============
T(n) = O(n * log(n))
```

2.

```
// ===================
// Recurrence Relation
// ===================
T(n) = 4T(n/2) + n^2


// ==============
// Master Theorem
// ==============
// T(n) = aT(n/b) + c*n^k
// T(1) = c
// a = 4, b = 2, c = 1, k = 2
// T(1) = 1
// b^k = 2^2 = 4
// a = 4
// CASE 2: a = b^k; T(n) = O(n^k * log(n))
// T(n) = O(n^2 * log(n))


// ===============
// Time Complexity
// ===============
T(n) = O(n^2 * log(n))
```

3.

```
// ===================

// Recurrence Relation

// ===================

T(n) = 8T(n/2) + 1


// ==============

// Master Theorem

// ==============

// T(n) = aT(n/b) + c*n^k

// T(1) = c

// a = 8, b = 2, c = 1, k = 1

// T(1) = 1

// b^k = 2^1 = 2

// a = 8

// CASE 3: a > b^k

// T(n) = O(n^log_2(8)) = O(n^3)


// ===============

// Time Complexity

// ===============

T(n) = O(n^3)
```

4.

```
// ===================
// Recurrence Relation
// ===================
T(n) = 2T(n/2) + sqrt(n)

// ==============
// Master Theorem
// ==============
// T(n) = aT(n/b) + c*n^k
// T(1) = c
// a = 2, b = 2, c = 1, k = 1/2
// T(1) = 1
// b^k = sqrt(2) = 2^(1/2)
// a = 2
// CASE 3: a > b^k; T(n) = O(n^log_b(a))
// T(n) = O(n^log_2(2)) = O(n)

// ===============
// Time Complexity
// ===============
T(n) = O(n)
```

5.

```
// ===================
// Recurrence Relation
// ===================
T(n) = 9T(n/3) + n


// ==============
// Master Theorem
// ==============
// T(n) = aT(n/b) + c*n^k
// T(1) = c
// a = 9, b = 3, c = 1, k = 1
// T(1) = 1
// b^k = 3^1 = 3
// a = 9
// CASE 3: a > b^k; T(n) = O(n^log_b(a))
// T(n) = O(n^log_3(9)) = O(n^2)


// ===============
// Time Complexity
// ===============
T(n) = O(n^2)
```

6.

```
// ===================
// Recurrence Relation
// ===================
T(n) = T(2n/3) + 1


// ==============
// Master Theorem Case
// ==============
// T(n) = aT(n/b) + c*n^k
// a = 1, b = 2/3, c = 1, k = 0
// T(1) = 1
// b^k = 1
// a = 1
// Case 2: a = b^k; T(n) = O(n^k * log(n))
// T(n) = O(n^0 * log(n)) = O(log(n))


// ===============
// Time Complexity
// ===============
T(n) = O(log(n))
```