## 1. Section 8 Exercises

**Program Memory**

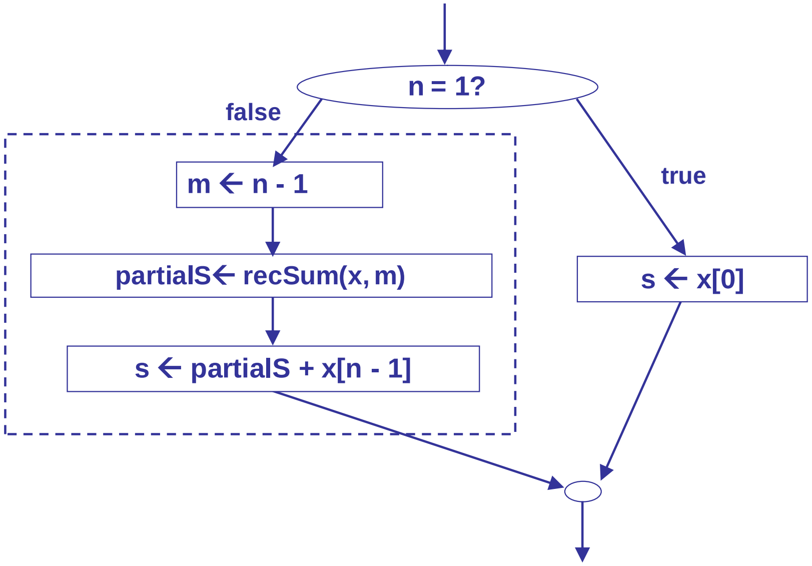
**Exercise 8-1** - Trace for this value of X

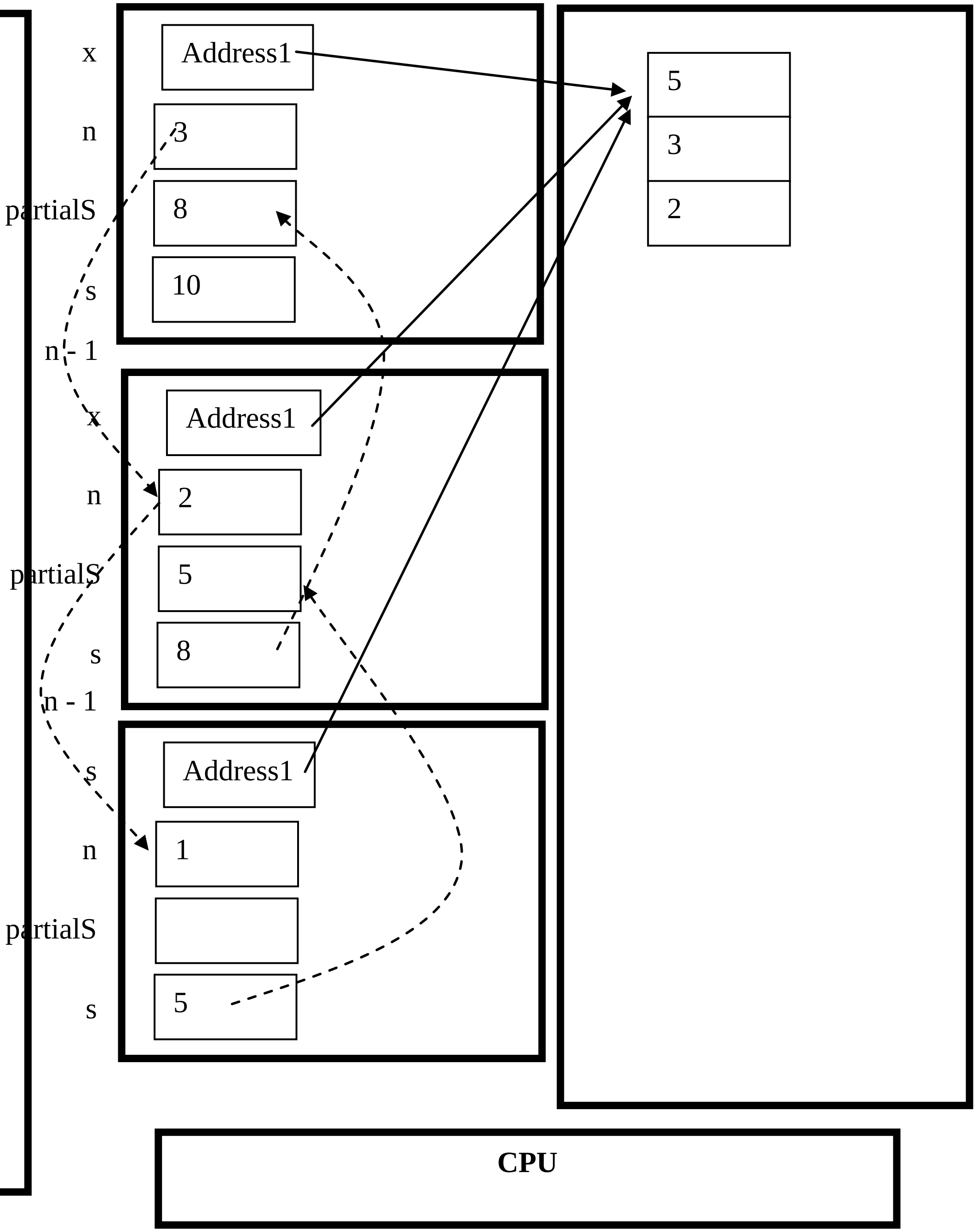
**Working memory**

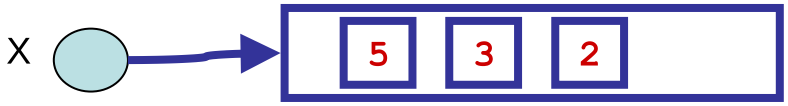
**Global Memory**

Trace call recSum(x, 3)

GIVENS: x (reference to an array of integers)
n (number of elements to sum in the array)
RESULT: s (sum of n elements in the array)
INTERMEDIATES:
partialS (partial sum of first n-1 elements in x)
HEADER:
s ← recSum(x, n)

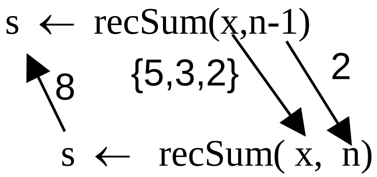






Exercise 8-1 – Trace, Table 1 – recSum(x, 3)

| statement | Ref. by x | n | partialS | s |
|---|---|---|---|---|
| initial values | { 5, 3, 2 } | 3 | ? | ? |
| 1. test: n = 1? false | | | | |
| 2. partialS ← recSum(x,n-1)<br>see Table 2 | | | 8 | |
| 3. s ← partialS + x[n-1] | | | | 10 |



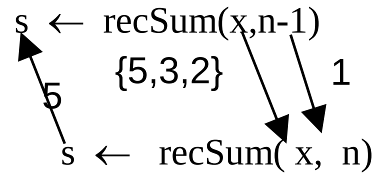
s ← recSum(x,n-1)

8 {5,3,2} 2

s ← recSum( x, n)

Exercise 8-1 – Trace, Table 2 – recSum(x, 2)

| statement | Ref. by x | n | partialS | s |
|---|---|---|---|---|
| initial values | { 5, 3, 2 } | 2 | ? | ? |
| 1. test: n = 1? false | | | | |
| 2. partialS ← recSum(x,n-1)<br>see Table 3 | | | 5 | |
| 3. s ← partialS + x[n-1] | | | | 8 |



s ← recSum(x,n-1)

5 {5,3,2} 1

s ← recSum( x, n)

Exercise 8-1 – Trace, Table 3 – recSum(x, 1)

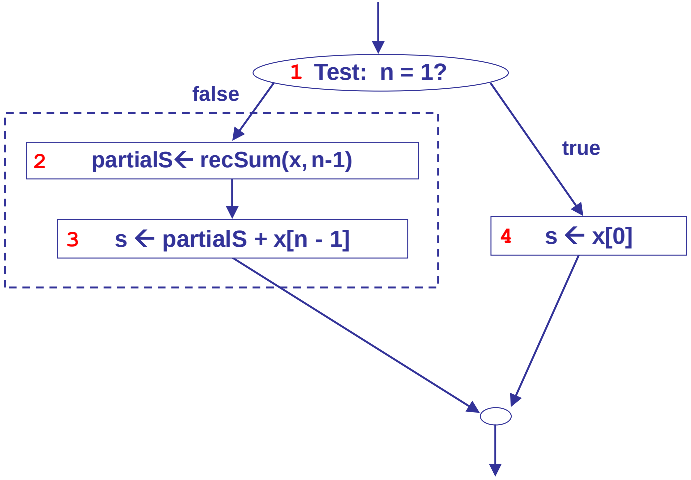
| statement | Ref. by x | n | partialS | s |
|---|---|---|---|---|
| initial values | { 5, 3, 2 } | 1 | ? | ? |
| 1. test: N = 1? true | | | | |
| 4. s ← X[0] | | | | 5 |

Program Memory

# Exercise 8-2 – Translating recSum to Java

Working memory Global Memory

GIVENS: x (a reference to an array of integers)
n (number of elements to sum in the array)
RESULT: x (sum of n elements in the array)
INTERMEDIATES:
partialS (partial sum of first N-1 elements in the array)
HEADER:
s ← recSum(x, n)

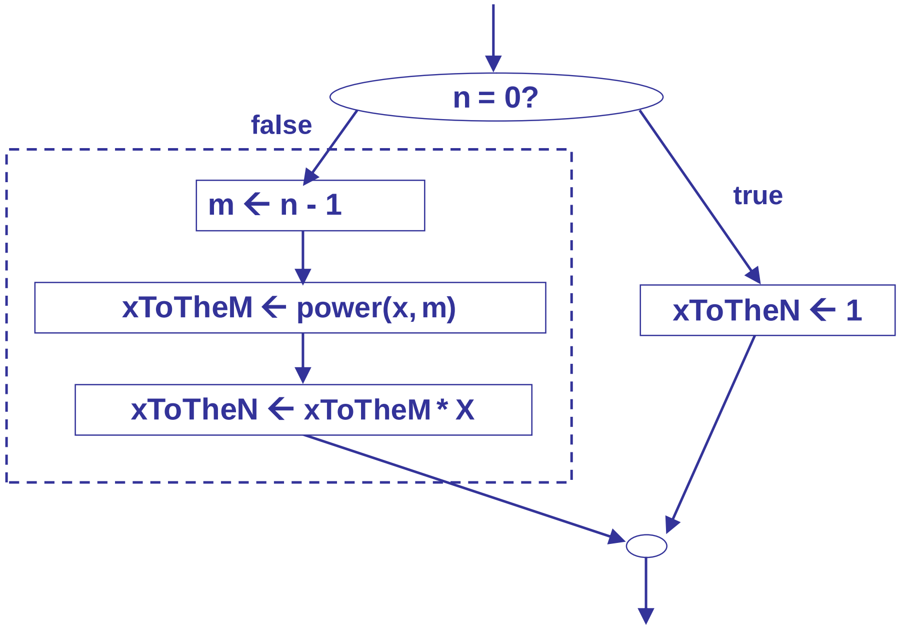


```
public static int recSum(int [] x,
                         int n)
{
   int partialS;
   int s;
   if (n == 1)
   {
      s = x[0];
   }
   else
   {
      partialS = recSum(x, n-1);
      s = partialS + x[n - 1];
   }
   return sum;
}
```

CPU

| Algorithm Model | Java |
|---|---|
| **Exercise 8-3 (a)** – Find $x^n$ where x and n are integers and $n \ge 0$, $x \ge 1$. (Version 1) | |

GIVENS: x *(base of exponentiation)*
n *(power to which x is raised)*
RESULT: xToTheN *(x to the power of n)*
INTERMEDIATES:
m *(set to n-1; smaller!)*
xToTheM *(partial results)*
HEADER: xToTheN ← *power (x,n)*
BODY:

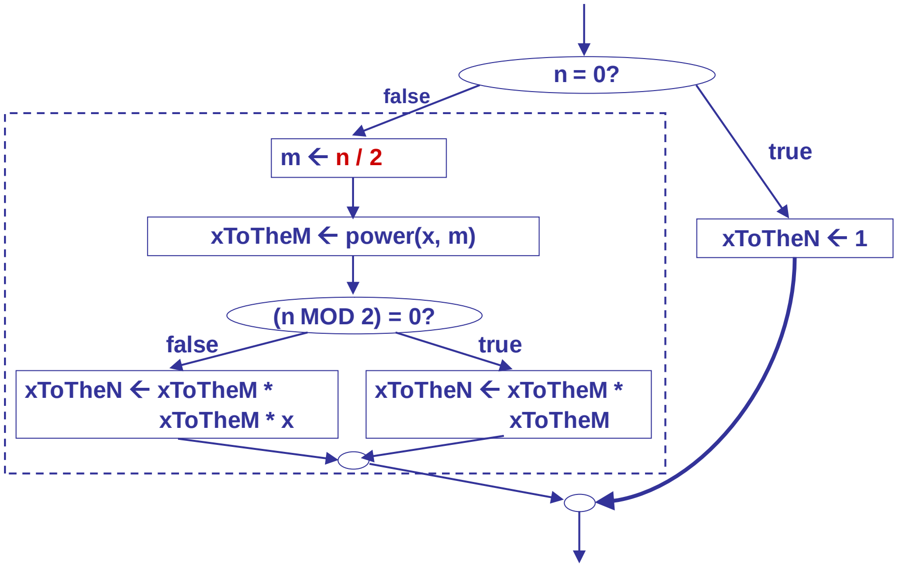


```
// METHOD power: find x to the power n
public static int power(int x, int n)
{
   // VARIABLE DECLARATION / DATA DICTIONNARY
      int m;      // INTERMEDIATE: reduced value
      int xToTheM;  // INTERMEDIATE: partial result
      int xToTheN;  // RESULT: expected result

   // ALGORITHM BODY
   if (n == 0)
   {
      xToTheN = 1;
   }
   else
   {
      m = n-1;
      xToTheM = power(x, m);
      xToTheN = xToTheM * x;
   }

   // RETURN RESULT
   return xToTheN;
}
```

| Algorithm Model | Java |
|---|---|

**Exercise 8-3 (b)** – Find $x^n$ where x and n are integers and $n \geq 0$, $x \geq 1$. (Version 2 – efficient version)

GIVENS: x *(base of exponentiation)*
n *(power to which x is raised)*
RESULT: xToTheN *(x to the power of n)*
INTERMEDIATES:
m *(set to n/2; smaller!)*
xToTheM *(partial results)*
HEADER: xToTheN ← power (x,n)
BODY:

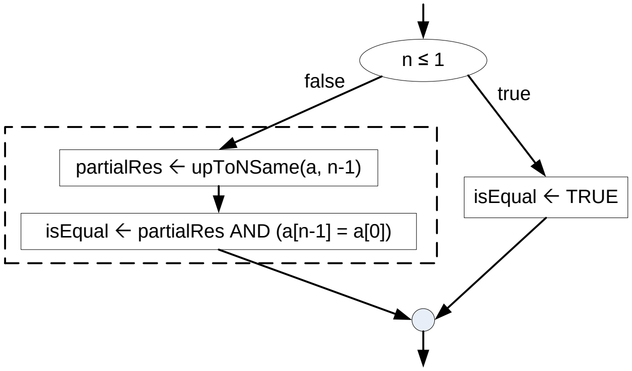


```
// METHODE power: find x to the power n
public static int power(int x, int n)
{
   // VARIABLE DECLARATION / DATA DICTIONNARY
      int m;      // INTERMEDIATE: reduced value
      int xToTheM;  // INTERMEDIATE: partial result
      int xToTheN;  // RESULT: expected result
   // ALGORITHM BODY
   if (n == 0)
   {
      xToTheN = 1;
   }
   else
   {
      m = n / 2;
      xToTheM = power(x, m);
      if ( n%2 == 0)
      {
         xToTheN = xToTheM * xToTheM;
      }
      else
      {
         xToTheN = xToTheM * xToTheM * x;
      }
   }
   // RETURN RESULT
   return xToTheN;
}
```

| Algorithm Model | Java |
|---|---|

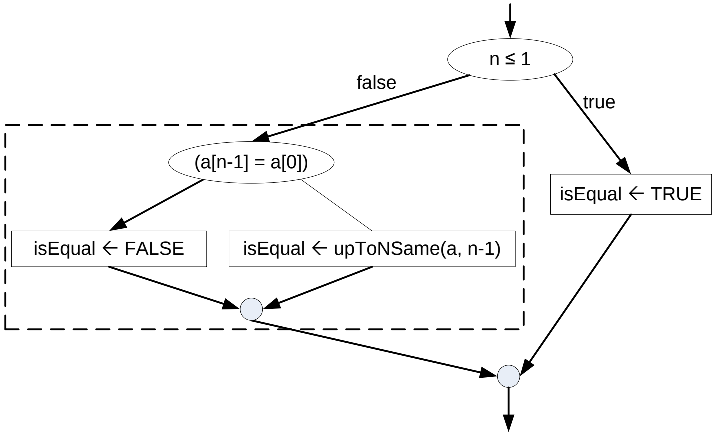
**Exercise 8-4** – Given an array a of more than n numbers, return TRUE if all the numbers in positions 0…n of a are equal, and false otherwise..

GIVENS: a *(An array of numbers)*
n *(Number of array elements to test)*
RESULT: isEqual *(Boolean: TRUE if all first N values in elements are equal)*
INTERMEDIATE:
partialRes *(partial result)*
HEADER: isEqual ← upToNSame(a, n)
BODY:



n ≤ 1
false
true
partialRes ← upToNSame(a, n-1)
isEqual ← partialRes AND (a[n-1] = a[0])
isEqual ← TRUE

Efficient Version

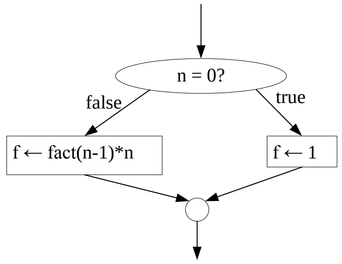


```
// Method upToNSame - Are elements up to n in a equal?
public static boolean upToNSame(int[] a, int n)
{
   // VARIABLE DECLARATION / DATA DICTIONNARY
      boolean partialRes;  // INTERMEDIATE: partial result
      boolean isEqual;   // RESULT: expected result
   // ALGORITHM BODY
   if (n <= 1)
   {
      isEqual = true;
   }
   else
   {
      partialRes = upToNSame(a, n-1);  // m=n-1 implicite
      isEqual = partialResult && (a[n-1] == a[0]);
   }
   // RETURN RESULT
   return isEqual;
}

// Method upToNSame - efficient version
public static boolean upToNSame(int[] a, int n)
{
   // VARIABLE DECLARATION / DATA DICTIONNARY
      boolean isEqual;   // RESULT: expected result
   // ALGORITHM BODY
   if (n <= 1)
   {
      isEqual = true;
   }
   else
   {
      if(a[n-1] == a[0])
            isEqual = upToNSame(a, n-1);
      else
            isEqual = false;
   }
   // RETURN RESULT
   return isEqual;
}
```

**Algorithm Model** | **Java**

## Exercise 8-5 – Calculate N !.

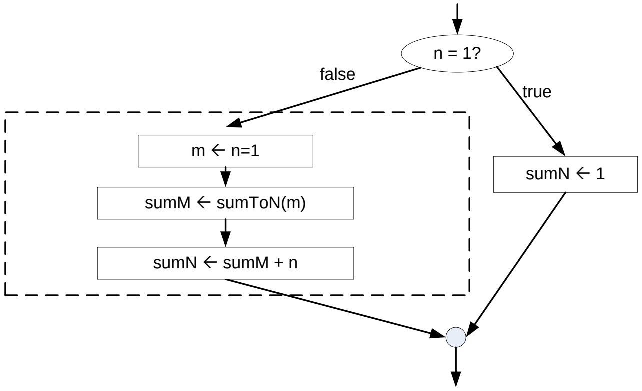
Givens: n *(integer)*
Results: f *(integer, n factorial)*
Intermediates: *(none)*
Header: f ← fact(n)
Body:



```
// Method fact
// Given: n, an integer
public static int fact(int n)
{
      int f;  // RESULT
      if (n == 0)
      {
        f = 1;
      }  // base case
      else
      {
        f = fact(n-1) * n;
      }
      return f;
}
```

## Exercise 8-6 – Find the sum of 1+2+…+N.

GIVENS: n *(An integer)*
RESULT: sumN *(sum of integers from 1 to n)*
INTERMEDIATE:
m *(set to n-1; smaller!)*
sumM *(sum of integers from 1 to m)*
HEADER: sumN ← sumToN(n)
BODY:



```
    public static int sumToN(int n)
    {
        // Variable Declarations
        int sumN;  // RESULT
        int m;     // INTERMEDIATE
        int sumM;  // INTERMEDIATE
        if(n == 1)
        {
            sumN = 1;  // base case
        }
        else
        {
            m = n-1;
            sumM = sumToN(M);  // recursive call
            sumN = sumM + 1;
        }
       // Return results
       return(sumN);
    }
```

| Algorithm Model | Java |
|---|---|

**Exercise 8-7** – Given an array A of N characters, reverse the values stored in positions Start to Finish.

GIVENS: a *(reference to a char. array to reverse)*<br>
low *(low index )*<br>
high *(high index )*

RESULTS: *(none)*

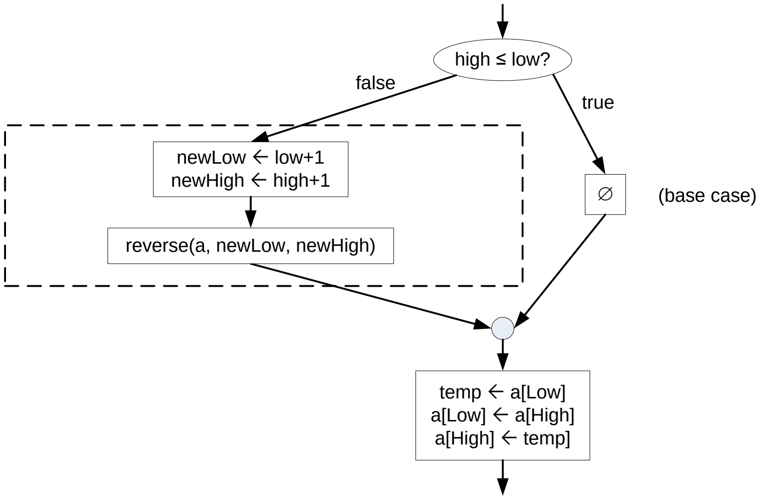
MODIFIED: a *(referenced array content changes)*

INTERMEDIATES:

newHigh *(new high index)*<br>
newLow *(new low index)*<br>
temp *(used for swapping)*

HEADER reverse(a, low, high)

BODY:

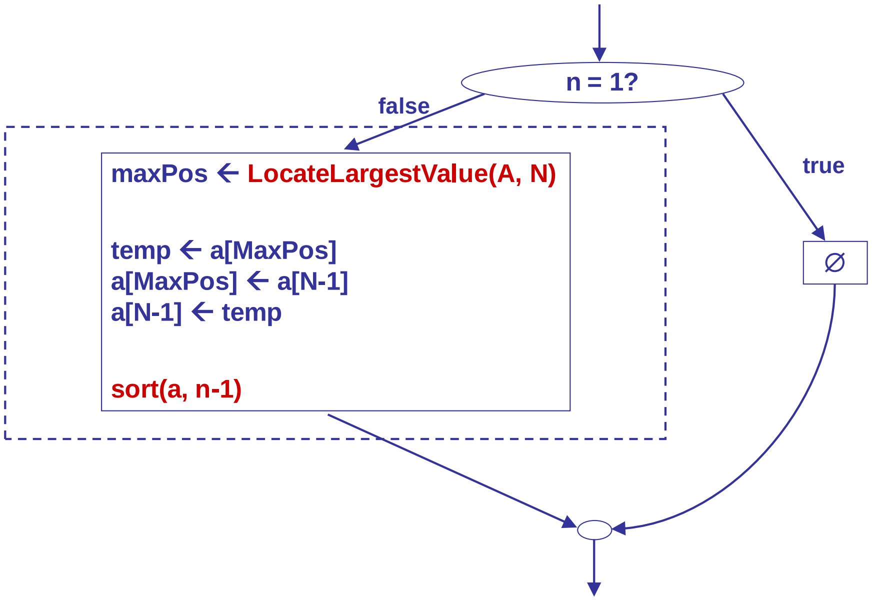


```
// Method : reverse: Reverse the characters in array a with
// size n.
// To be called initially with reverse(a, 0, n-1)
public static void reverse(char [] a, int low, int high)
{
   // VARIABLE DECLARATION / DATA DICTIONNARY
   int newHigh;  // INTERMEDIATE: smaller high
   int newLow;   // INTERMEDIATE: greater low
   char temp;     // INTERMIDATE:  buffer for char
   // ALGORITHM BODY
   if (high - low <= 1)
   {
      /* base case: do nothing */ ;
   }
   else
   {
      newlow = low + 1,
      newHigh = high - 1;  // 2 variables to make "smaller"!
      reverse(a, newLow, newHigh);
   }
   // reverse a[low] and a[high]
   temp = a[low];
   a[low] = a[high];
   a[high] = temp;
   // RESULT: The array reference by 'a' is modified!
}
```

**Algorithm Model** | **Java**

**Exercise 8-8** – Sort an array of numbers in increasing order: – sort algorithm/method

GIVENS: a *(reference to an array to sort)*
n *(number of elements in array)*
RESULTS: *(none)*
MODIFIED: a *(sorted array)*
INTERMEDIATES:
maxPos *(position of largest value in array)*
temp *(used for swapping)*
HEADER sort( a, n )
BODY:

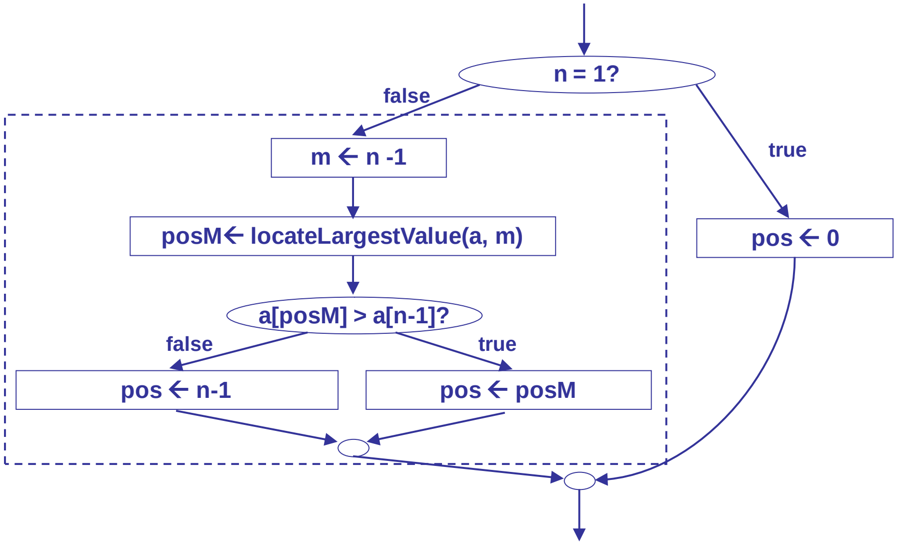


```
public static void sort(int[] a, int n)
{
  // VARIABLE DECLARATIONS
  // GIVENS: a - reference to array to sort
  //         n - number of elements to sort - note
  //             that a.length CANNOT be used.
  // INTERMEDIATES
  int maxPos;  // position of largest value
  int temp; // used for swapping
  // BODY
  if(n <= 1)
  {
      /* do nothing */ ;
  }
  else
  {
      maxPos = locateLargestValue(a, n);
      temp = a[maxPos];
      a[maxPos] = a[n-1];
      a[n-1] = temp;
      sort(a,n-1);  // sort rest of array
  }
}
```

**Algorithm Model** | **Java**

**Exercise 8-8** – Sort an array of numbers in increasing order: – locateLargestValue algorithm/method

```
GIVENS:  a      (an array to sort)
         n      (number of elements in array)
RESULTS: pos    (index of largest value in array)
INTERMEDIATES:
         m      (integer, smaller interval)
         posM       (position of largest value in
                     smaller array)
HEADER  pos ← LocateLargestValue( a, n )
BODY:
```



```java
public static void locateLargestValue(int[] a, int n)
{
  // VARIABLE DECLARATIONS
  // GIVENS: a - reference to array to sort
  //         n - number of elements to sort - note
  //             that a.length CANNOT be used.
  int pos;  // RESULT - position of largest value
  // INTERMEDIATES
  int m;    // smaller
  int posM; // used for swapping

  if(n == 1)
  {
     pos = 0;  // base case
  }
  else
  {
     m = n-1;
     posM = locateLargestValue(a, m);  // recursion
     if(a[posM] > a[n-1])
     {
       pos = posM;
     }
     else
     {
       pos = n-1;
     }
  }
  // RESULT
  return(pos);
}
```