Data Structures and Algorithms
Lecture slides: Programming languages implementations of recursion

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

Topics cover

Activation records & run time stacks

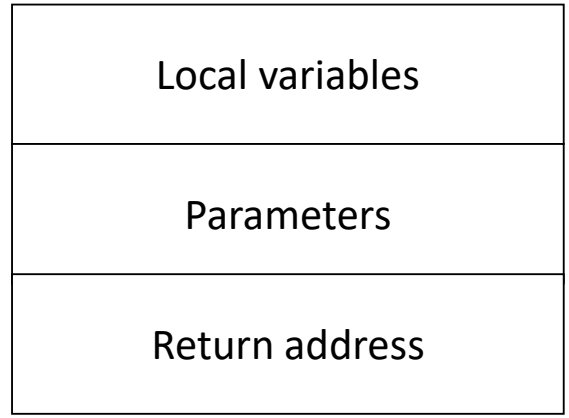
Computing factorial

Binary search

Merge sort

Activation record

- An activation record is a data structure that contains the following fields among others:
  - Local variables
  - Function parameters
  - A return address



An activation record is created each time a function call is made in a program

The first activation record is the one corresponding to the main

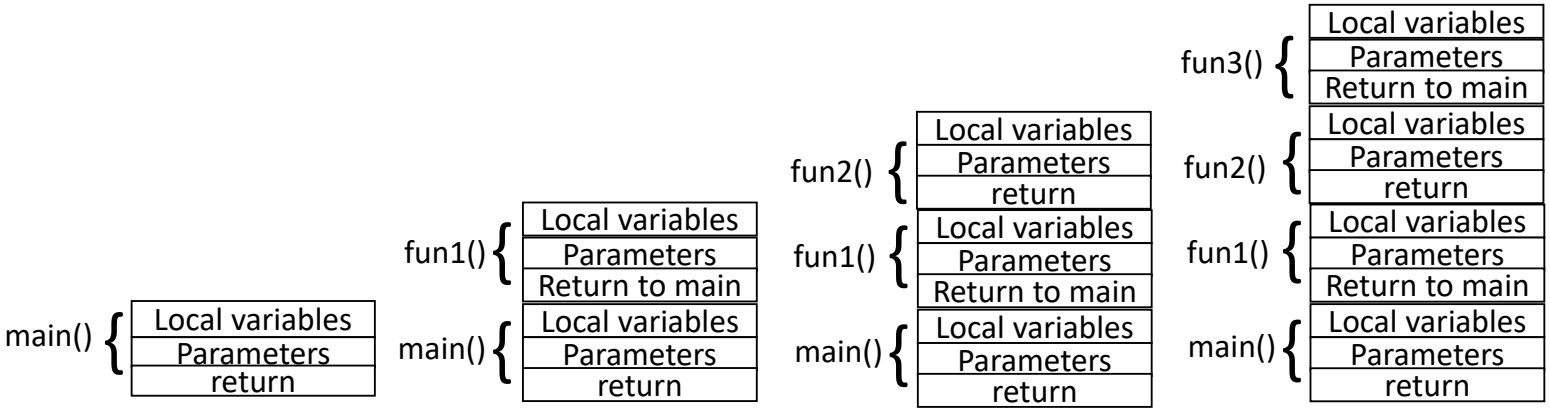
Activation record

- This program has 3 function calls:
  - main() calls fun1()
  - fun1() calls fun2()
  - fun2() calls fun3()
- Consequently 4 activation records will be created at run time
  - One once the program starts to run
  - A second when main arrives at the instruction that calls fun1()
  - Third when fun1() arrives at the instruction that calls fun2()
  - Finally a fourth one once fun2() arrives at the instruction that call fun3()

```
void fun1(float r) {
        int s, t;
          ...
        fun2(s);
          ...
        }
void fun2(int x) {
         int y;
          ...
        fun3(y);
          ...
        }
void fun3(int q) {
          ...
        }
  void main() {
        float p;
          ...
        fun1(p);
          ...
        }
```

4 activation records

- The four activation records created as function calls are made consecutively



Function calls

- A function call causes the program to jump to the first instruction of the function it calls
- When main calls fun1(), the program stops executing instructions in main and go to execute instructions in fun1()
- Once a function has executed all its instructions, the program returns to execute instructions from the caller
- So once fun1() has completed, the program will continue to execute instructions in main, precisely the instruction just after the call to fun1()

```
void fun1(float r) {
        int s, t;
          ...
        fun2(s);
          ...
        }
void fun2(int x) {
         int y;
          ...
        fun3(y);
          ...
        }
void fun3(int q) {
          ...
        }
  void main() {
        float p;
          ...
        fun1(p);
          ...
        }
```

Function calls & activation records

- A program remember where to return at the end of the execution of a function because it has stored the address of the return instruction in the activation record of the called function
- For example, once fun1() has completed its execution, the return address in main can be found from the activation record of fun1()

```
void fun1(float r) {
        int s, t;
          ...
        fun2(s);
          ...
        }
void fun2(int x) {
         int y;
          ...
        fun3(y);
          ...
        }
void fun3(int q) {
          ...
        }
  void main() {
        float p;
          ...
        fun1(p);
          ...
        }
```

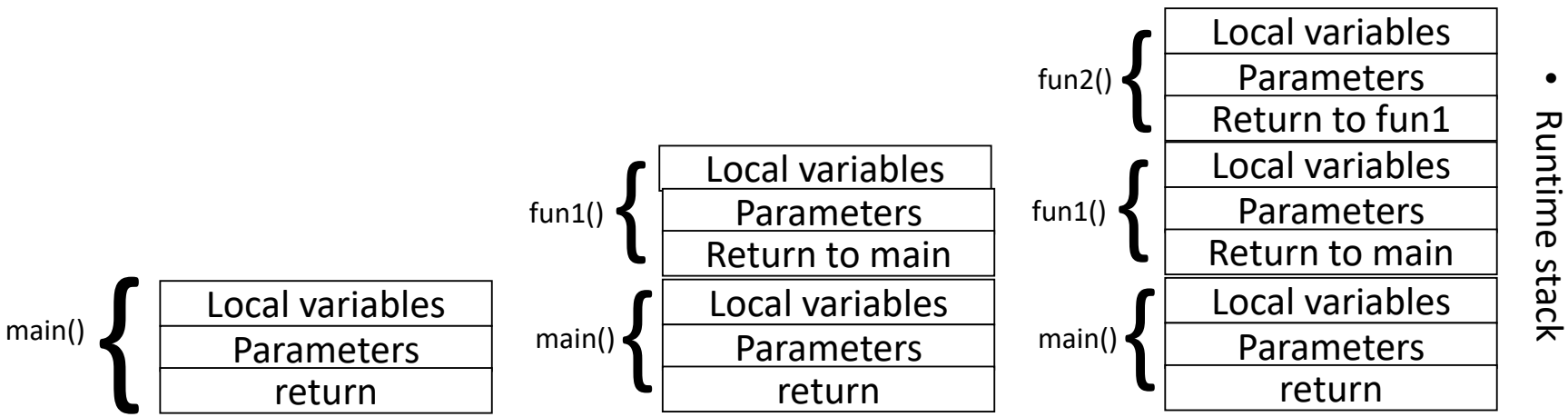
Activation records, function calls & runtime stack

- Not only a program remember to which instruction it should returns but it also knows to which function it should return thanks to the runtime stack that pile the activations as the are created
- This program has to execute 3 consecutive function calls before returning to main()
  - main() → fun1()
  - fun1() → fun2()
  - fun2() → fun3()
- Then
  - fun3() returns to fun2()
  - fun2() returns to fun1()
  - fun1() returns to main()
- The program has to remember all these function calls and returns, the runtime stack take care of this

```
void fun1(float r) {
        int s, t;
          ...
        fun2(s);
          ...
        }
void fun2(int x) {
         int y;
          ...
        fun3(y);
          ...
        }
void fun3(int q) {
          ...
        }
  void main() {
        float p;
          ...
        fun1(p);
          ...
        }
```

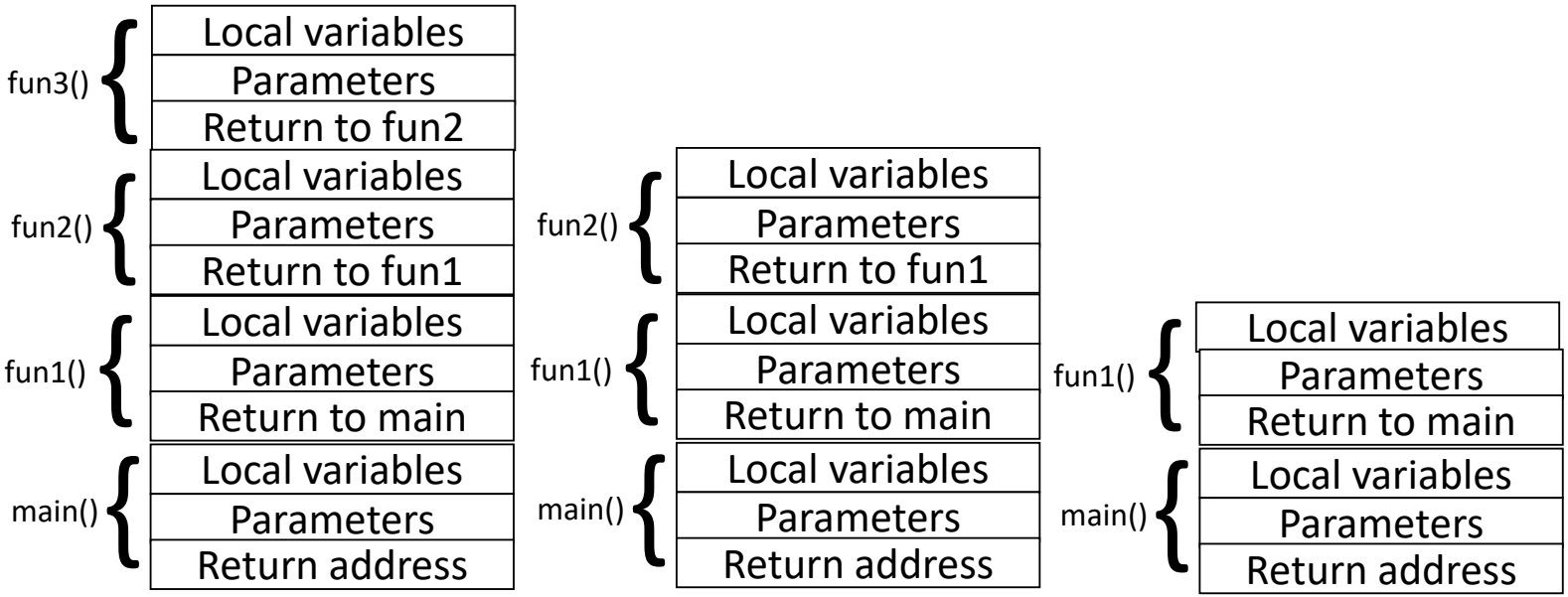
Runtime stack

- Here the program first calls fun1(), then fun1() call fun2() which then call fun3()
- Each call adds one activation record on top of the runtime stack



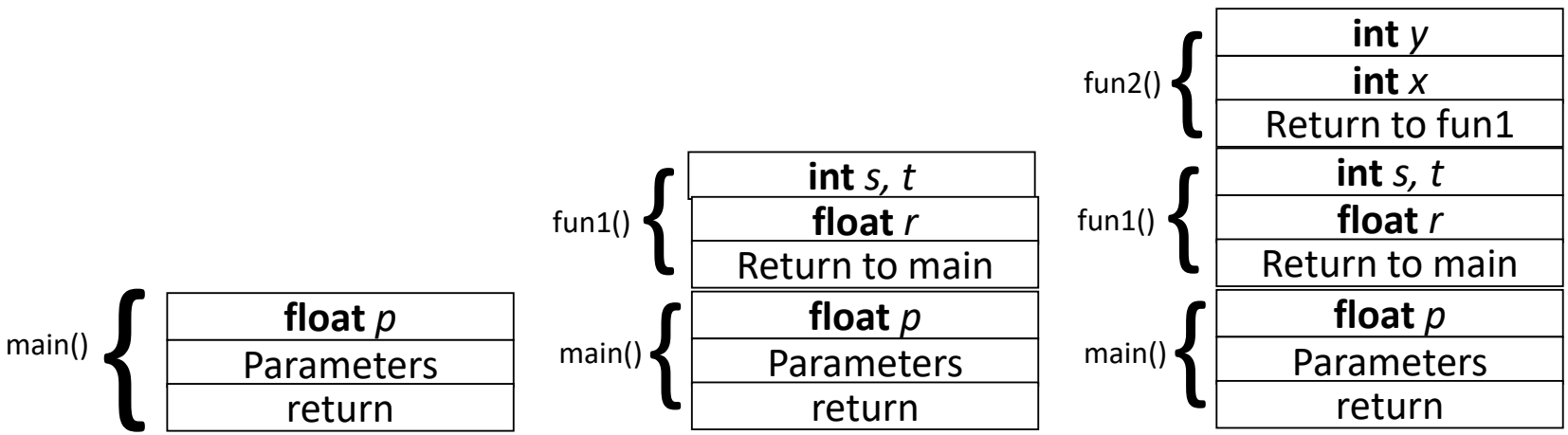
Runtime stack

- The function currently running is always the one for which the activation record is on top of the runtime stack
- Once that function returns
  - the corresponding activation record is popped out of the runtime stack
  - the function corresponding to the new top of the stack resumes execution



Runtime stack,function arguments and local variables

- Not only the return address is pushed on the stack, also the function arguments and local variables



Recursive function for factorial

A recursive function is one that makes a call to itself inside its own code

Example factorial : $n \times n-1 \times n-2 \times \ldots \times 2 \times 1$

Below is a recursive implementation of factorial n

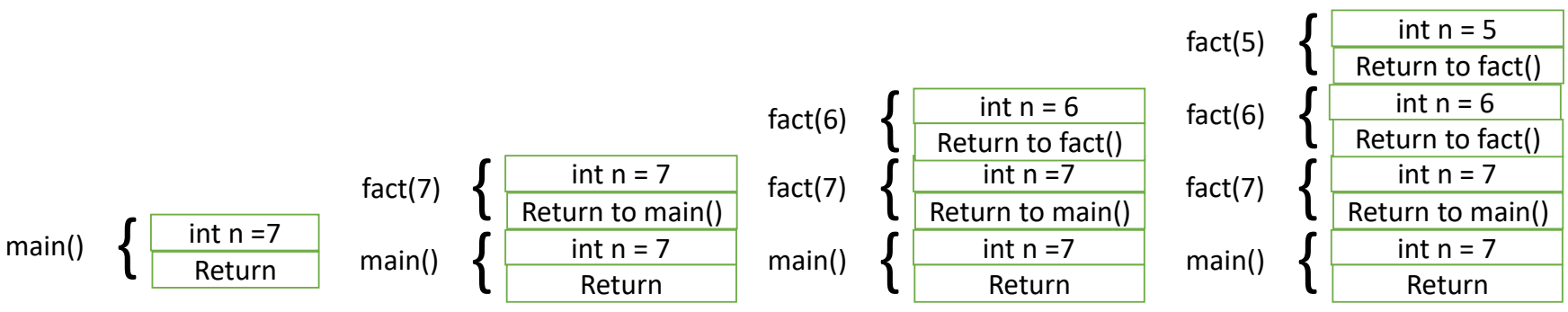
```
factorial (n)
        if (n ≤ 1)
          return 1 ;
        else
          return n × factorial(n − 1) ;
```

The function "factorial" is first called with n as argument, then it calls itself with argument $n-1$.

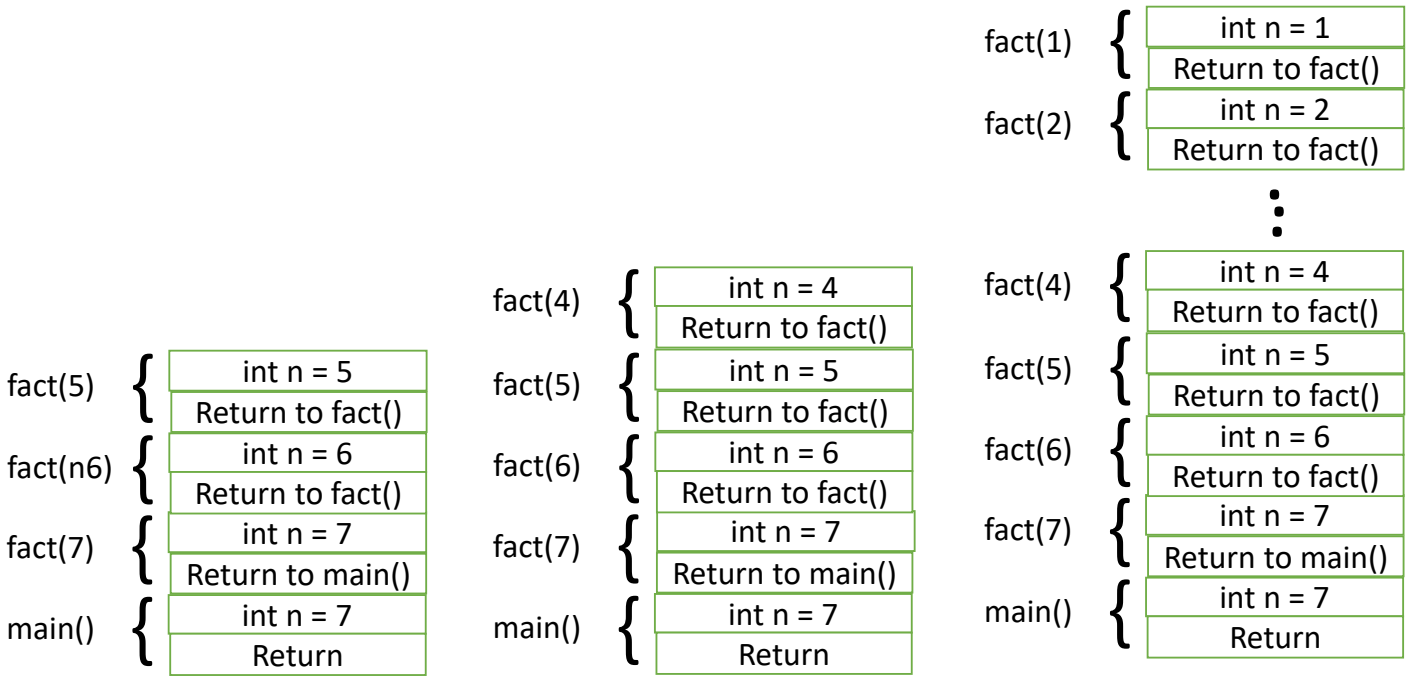
C code implementing recursive factorial

```
#include<stdio.h>
int fact(int n);
int main() {
        int n;
        printf("Enter a positive integer: ");
        scanf("%d",&n);
        printf("Factorial of %d = %ld", n, fact(n));
        return 0;
}
int fact(int n) {
        if (n<=1)
                return 1;
        else
                return n*fact(n-1);
}
```

Runtime stacks for n = 7



Runtime stacks for n = 7



Factorial 7

Entering factorial with n = 7
Entering factorial with n = 6
Entering factorial with n = 5
Entering factorial with n = 4
Entering factorial with n = 3
Entering factorial with n = 2
Entering factorial with n = 1
Exiting factorial for n = 1, factorial of n is 1
Exiting factorial for n = 2, factorial of n is 2
Exiting factorial for n = 3, factorial of n is 6
Exiting factorial for n = 4, factorial of n is 24
Exiting factorial for n = 5, factorial of n is 120
Exiting factorial for n = 6, factorial of n is 720
Exiting factorial for n = 7, factorial of n is 5040

A second example : recursive binary Search

Given a sorted array A of size n, and a searched value x, recursive binary search compares x with $A[\lfloor \frac{n}{2} \rfloor]$

- if $x < A[\lfloor \frac{n}{2} \rfloor]$, recursively searches the sub-array $A[1..\lfloor \frac{n}{2} \rfloor - 1]$
- if $x > A[\lfloor \frac{n}{2} \rfloor]$, recursively searches the sub-array $A[\lfloor \frac{n}{2} \rfloor + 1, n]$
- if $x = A[\lfloor \frac{n}{2} \rfloor]$, return $\lfloor \frac{n}{2} \rfloor$, the index in $A[]$ where x has been found

BinarySearch$(A[i..j], x)$

if $i > j$ **then return** -1
$k = \lfloor \frac{(i+j)}{2} \rfloor$
if $A[k] = x$ **return** k
if $x < A[k]$ **then**
  return $BinarySearch(A[i..k-1], x)$
else
  return $BinarySearch(A[k+1..j], x)$

C code implementing recursive binary search

```
#include<stdio.h>
int RecursiveBsearch(int A[], int i, int j, int x) {
   if(i>j) return -1;
   int mid = (i+j)/2;
   if( A[mid] == x ) return mid;
   else if( x < A[mid] )
      RecursiveBsearch(A, i, mid-1, x);
   else
      RecursiveBsearch(A, mid+1, j, x);
}
int main() {
   int A[] = 0,2,6,11,12,18,34,45,55,99;
   int x=55;
   printf("%d is found at Index %d",x,RecursiveBsearch(A,0,9,x));
   return 0;
}
```

Binary search : code tracing the recursive procedure

```
int RecursiveBsearch(int A[], int i, int j, int x) {
  if(i>j) {
              printf(“x not found, i = 2 > j = 1, returning -1\n”,i,j);
              return -1;
              }
    int mid = (i+j)/2;
    if( A[mid] == x ) {
              printf("Found the value, returning the value of index = %d\n", mid);
              return mid;
               }
  else if( x < A[mid] ){
               printf("x %d is smaller then A[mid] %d where mid is %d so calling left recursion\n", x, A[mid], mid);
               solution = RecursiveBsearch(A, i, mid-1, x);
                printf("Returning from left recursion of mid = %d with returned value = %d \n",mid,solution);
                  return solution;
                }
  else {
                   printf("x %d is greater then A[mid] %d where mid is %d so calling right recursion\n", x, A[mid], mid);
                 solution =  RecursiveBsearch(A, mid+1, j, x);
               printf("Returning from right recursion with mid = %d with returned value = %d \n",mid,solution);
                   return solution;
                }
}
```

Binary search : steps of the recursion

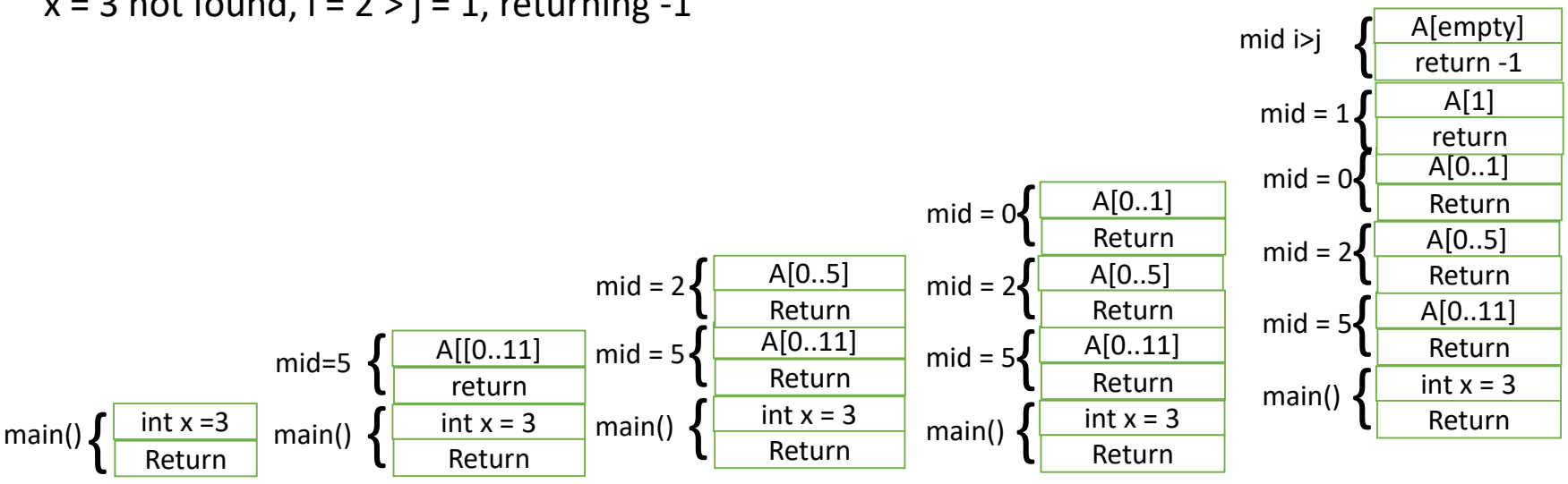
Binary search steps for $A = [0, 2, 6, 11, 12, 18, 34, 45, 55, 99, 103, 114]$ and $x = 3$

```
if (i > j) { printf("x = 3 not found, i > j, returning -1\n",i,j); return -1;}
else if( x < A[mid] ){
printf("x %d is smaller then A[mid] %d where mid is %d so calling left recursion\n", x, A[mid], mid);
solution = RecursiveBsearch(A, i, mid-1, x);
printf("Returning from left recursion of mid = %d with returned value = %d \n",mid,solution);
return solution;
}
else {
printf("x %d is greater then A[mid] %d where mid is %d so calling right recursion\n", x, A[mid], mid);
solution = RecursiveBsearch(A, mid+1, j, x);
printf("Returning from right recursion with mid = %d with returned value = %d \n",mid,solution);
return solution;
```

```
x = 3 is smaller then A[mid] 18 where mid is 5 so calling left recursion
x = 3 is smaller then A[mid] 6 where mid is 2 so calling left recursion
x = 3 is greater then A[mid] 0 where mid is 0 so calling right recursion
x = 3 is greater then A[mid] 2 where mid is 1 so calling right recursion
x = 3 not found, i = 2 > j = 1, returning -1
Returning from right recursion with mid = 1 with returned value = -1
Returning from right recursion with mid = 0 with returned value = -1
Returning from left recursion of mid = 2 with returned value = -1
Returning from left recursion of mid = 5 with returned value = -1
```

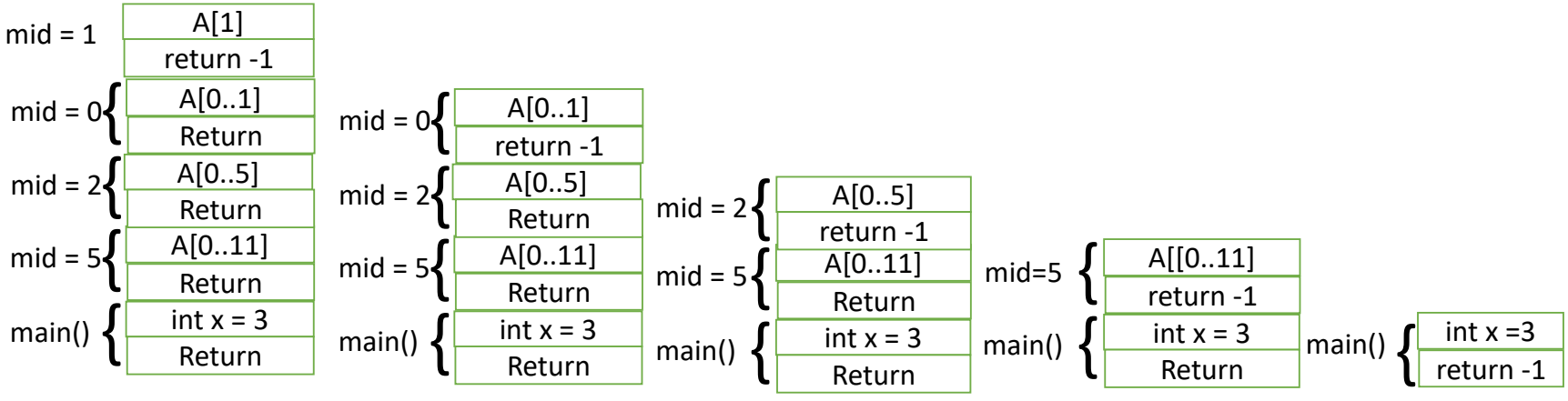
Binary search : run time stack, pushing activation records

x = 3 is smaller then A[mid] 18 where mid is 5 so calling left recursion
x = 3 is smaller then A[mid] 6 where mid is 2 so calling left recursion
x = 3 is greater then A[mid] 0 where mid is 0 so calling right recursion
x = 3 is greater then A[mid] 2 where mid is 1 so calling right recursion
x = 3 not found, i = 2 > j = 1, returning -1



Binary search : run time stack, popping activation records

Returning from right recursion with mid = 1 with returned value = -1
Returning from right recursion with mid = 0 with returned value = -1
Returning from left recursion of mid = 2 with returned value = -1
Returning from left recursion of mid = 5 with returned value = -1
3 is found at Index -1



Merge Sort

Merge sort is a recursive algorithm to sort an array of integers

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

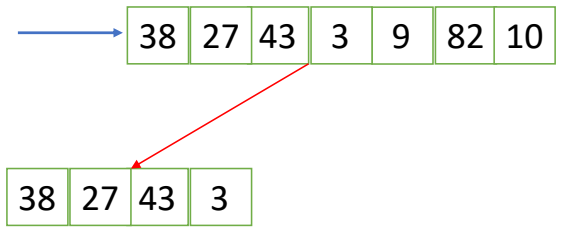
Merge Sort

The code below only describe the recursive part of merge sort

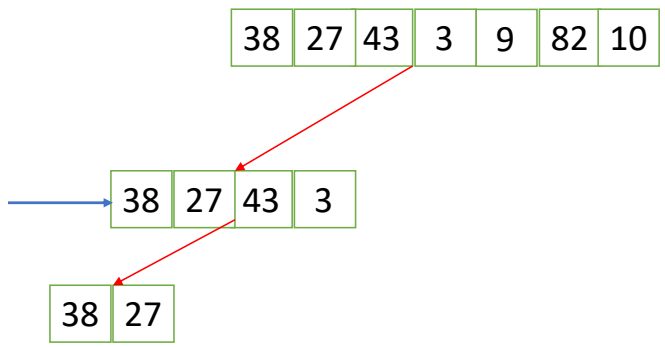
Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)

Merge Sort : execution sequence

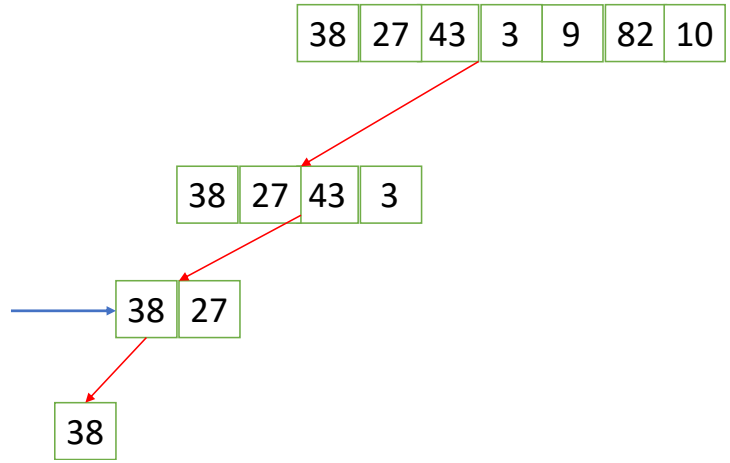
Mergesort($A[1..7]$)
  if $p < r$
    $4 = \lfloor \frac{1+7}{2} \rfloor$
→ Mergesort($A[1..4]$)
    Mergesort($A[q+1..r]$)



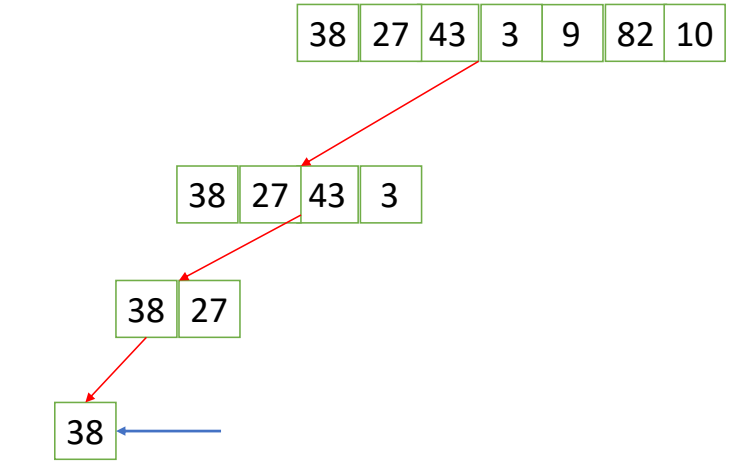
Mergesort($A[1..4]$)
  if $1 < 4$
    $2 = \lfloor \frac{1+4}{2} \rfloor$
→ Mergesort($A[1..2]$)
    Mergesort($A[q+1..r]$)



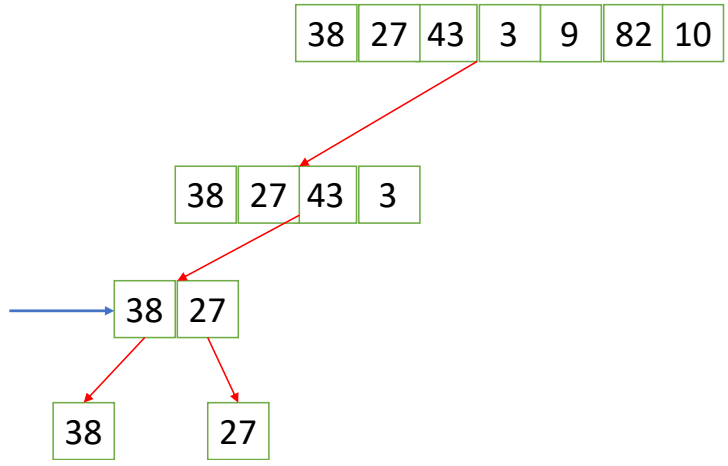
Mergesort($A[1..2]$)
  if $1 < 2$
    $q = \lfloor \frac{1+2}{2} \rfloor$
→ Mergesort($A[1..1]$)
    Mergesort($A[q+1..r]$)



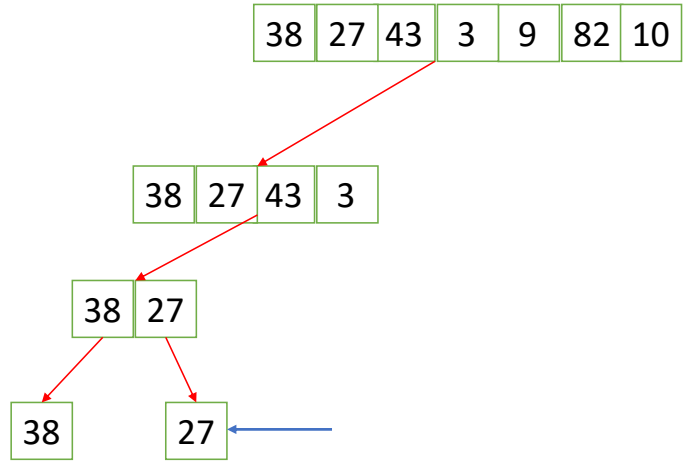
Mergesort($A[1..1]$)
→**if** $1 < 1$
    $q = \lfloor \frac{1+4}{2} \rfloor$
    Mergesort($A[1..2]$)
    Mergesort($A[q+1..r]$)



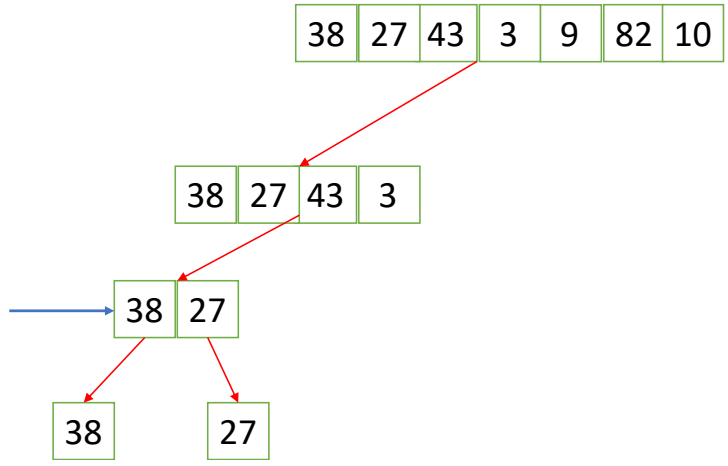
Mergesort($A[1..2]$)
  if $1 < 2$
    $1 = \lfloor \frac{1+2}{2} \rfloor$
    Mergesort($A[1..1]$)
→ Mergesort($A[2..2]$)



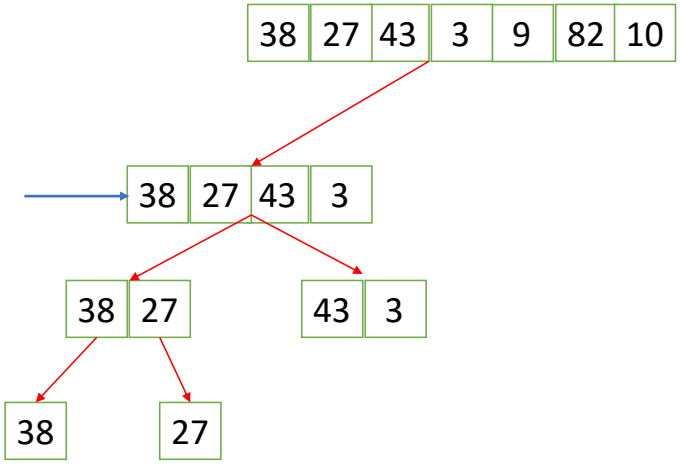
Mergesort($A[2..2]$)
→**if** $2 < 2$
    $q = \lfloor \frac{1+4}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[2..2]$)



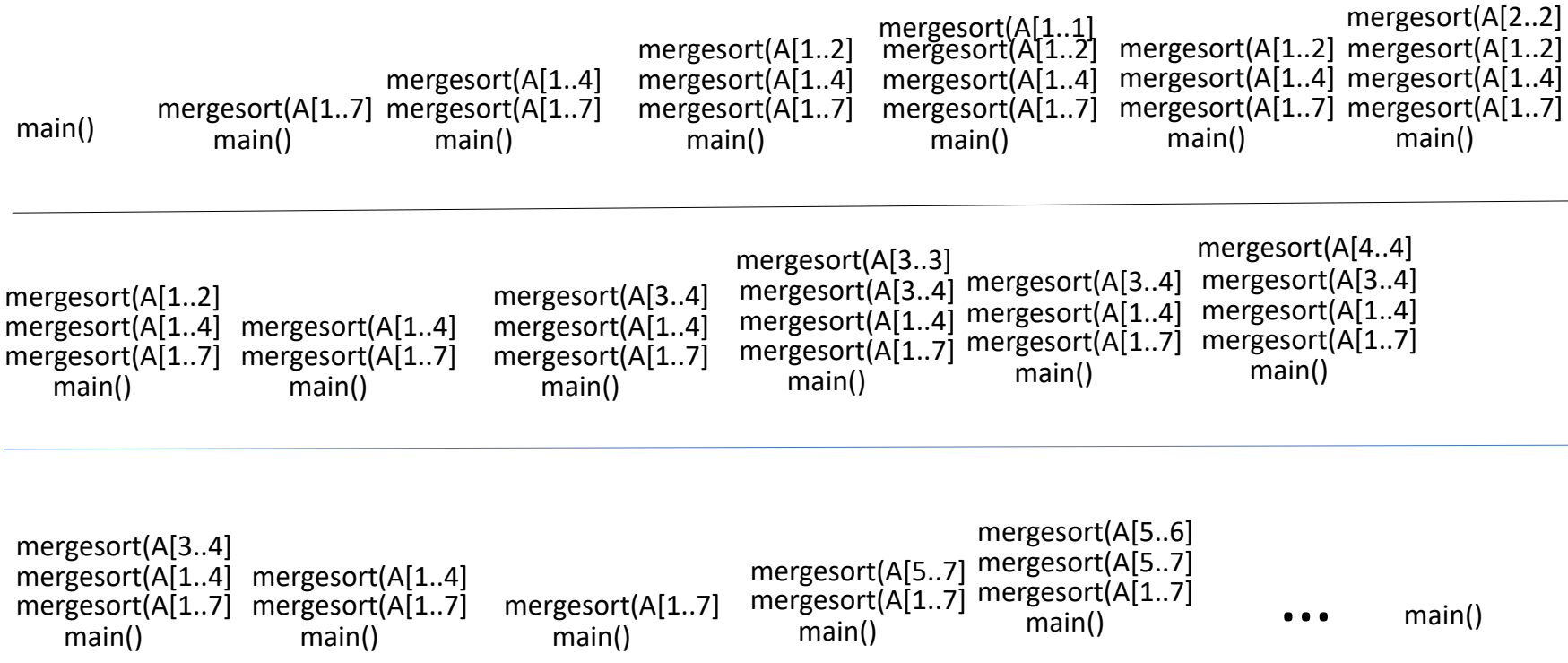
Mergesort($A[1..2]$)
  if $1 < 2$
    $q = \lfloor \frac{1+2}{2} \rfloor$
    Mergesort($A[1..1]$)
    Mergesort($A[2..2]$)
$\rightarrow$



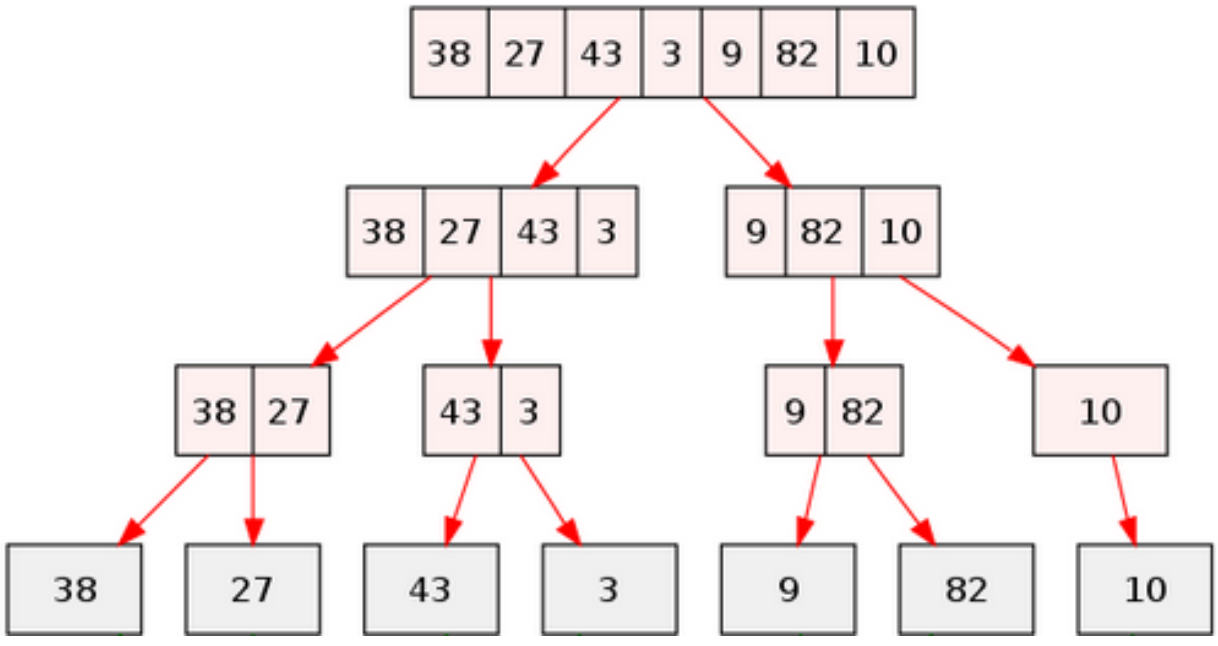
Mergesort($A[1..4]$)
  if $1 < 4$
    $q = \lfloor \frac{1+4}{2} \rfloor$
    Mergesort($A[1..2]$)
$\rightarrow$ Mergesort($A[3..4]$)



Merge sort in terms of activation records



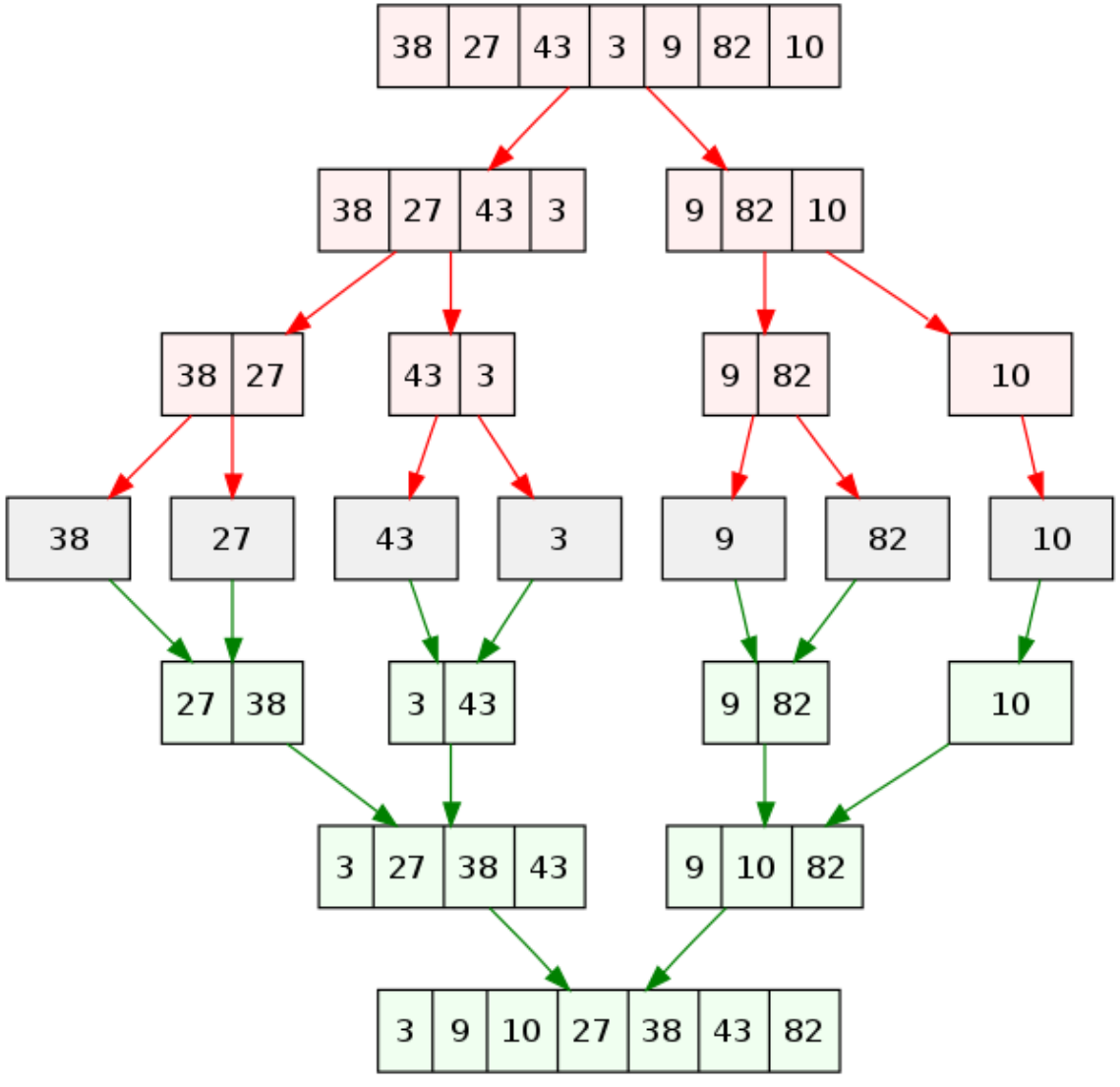
The complete "call tree"



The divide and merge parts !

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
→   Merge(A, p, q, r)

The merge part takes 2 sorted arrays and merges them into a single array sorted in increasing order



Merge algorithm

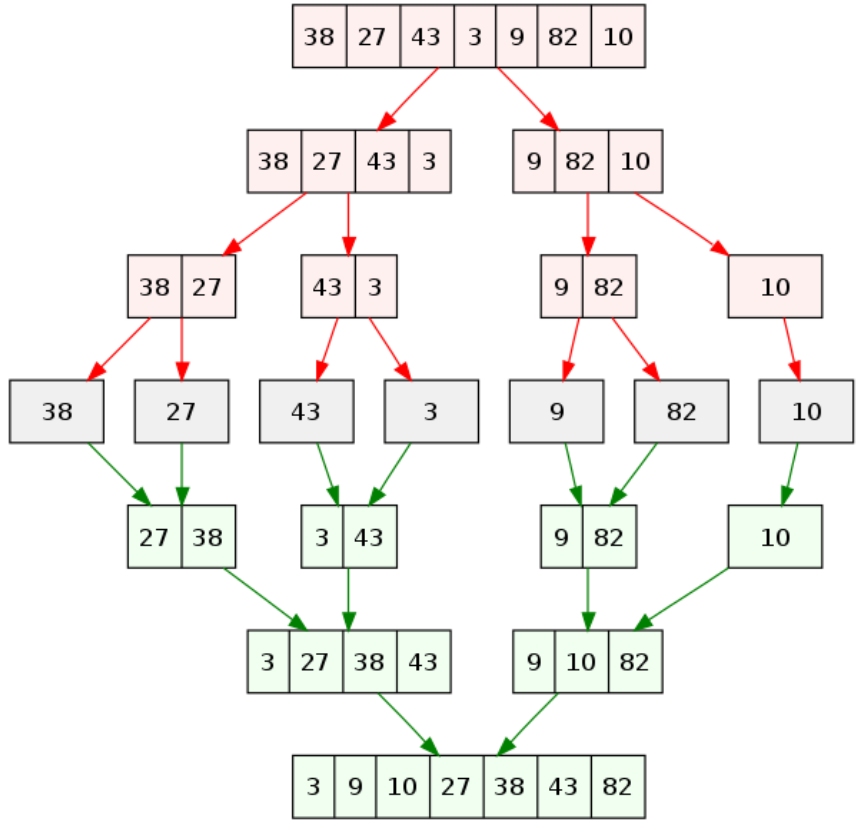
```
Merge(A, p, q, r)
n1 = q − p + 1
n2 = r − q
L[1..n1 + 1], R[1..n2 + 1]
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = −∞
R[n2 + 1] = −∞
i = 1, j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
```

Works as follows :

- Has a pointer to the beginning of each subarray.
- Put the smaller of the elements pointed to in the new array.
- Move the appropriate pointer.
- Repeat until new array is full.

Merge algorithm

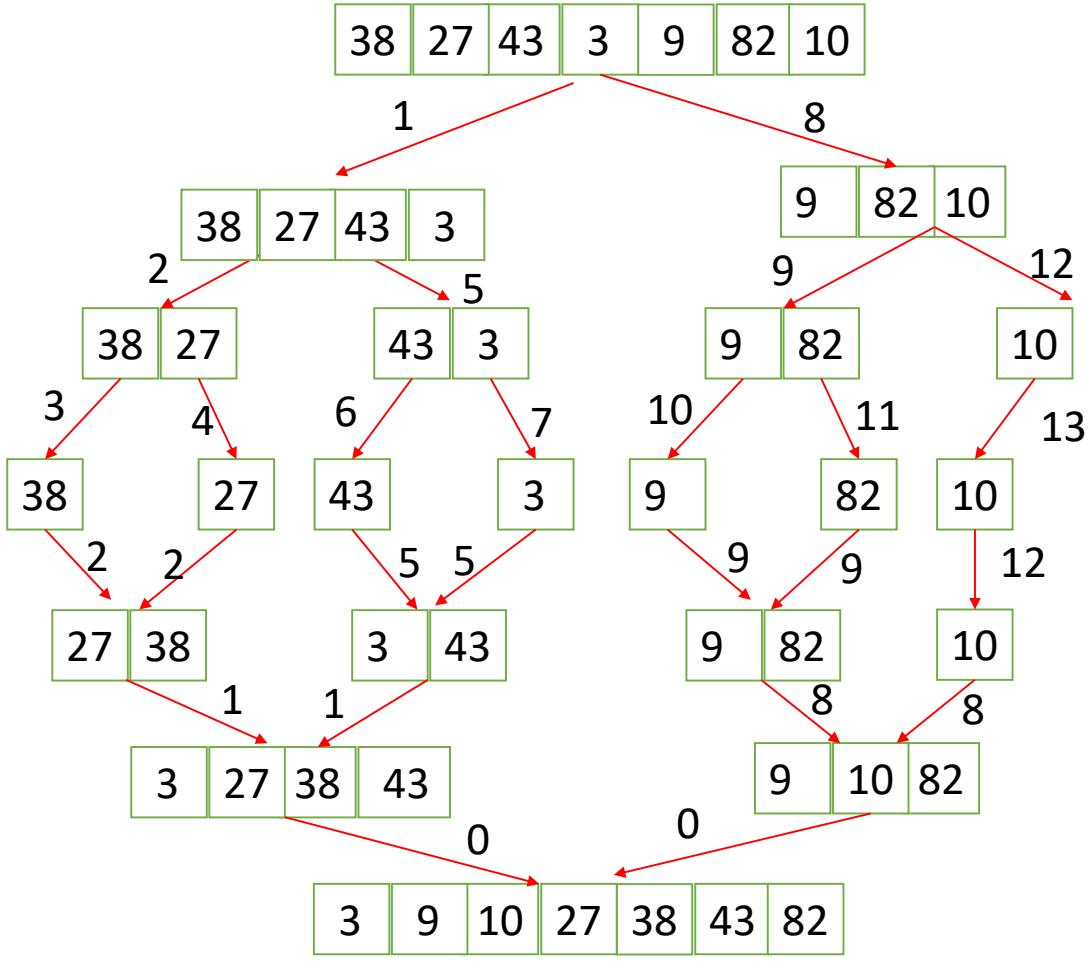
$Merge(A, p, q, r)$
$n1 = q - p + 1$
$n2 = r - q$
$L[1..n1 + 1]$, $R[1..n2 + 1]$
for $i = 1$ **to** $n1$
  $L[i] = A[p + i - 1]$
for $j = 1$ **to** $n2$
  $R[j] = A[q + j]$
$L[n1 + 1] = -\infty$
$R[n2 + 1] = -\infty$
$i = 1, j = 1$
for $k = p$ **to** r
  if $L[i] \leq R[j]$
    $A[k] = L[i]$
    $i = i + 1$
  else
    $A[k] = R[j]$
    $j = j + 1$



Sequence of recursive calls

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
L- Mergesort($A[p..q]$)
R- Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

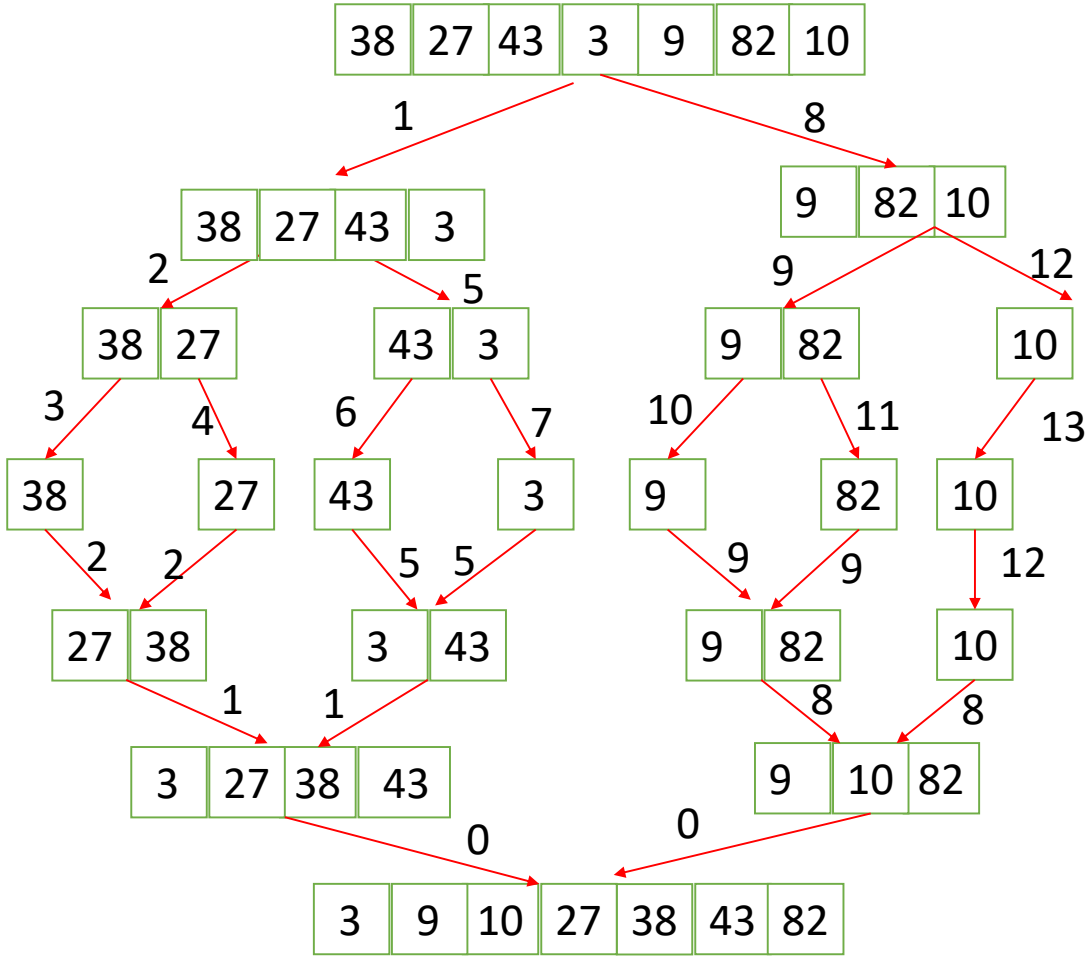
1 L-Mergesort(A[1..4])
2 L-Mergesort(A[1..2])
3 L-Mergesort(A[1..1])
4 R-Mergesort(A[2..2])
2 Merge(A,1,1,2)
5 R-Mergesort(A[3..4])
6 L-Mergesort(A[3..3])
7 R-Mergesort(A[4..4])
5 Merge(A,3,3,4)
1 Merge(A,1,2,4)



Sequence of recursive calls

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
L- Mergesort($A[p..q]$)
R- Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

8 R-Mergesort(A[5..7])
9 L-Mergesort(A[5..6])
10 L-Mergesort(A[5..5])
11 R-Mergesort(A[6..6])
9 Merge(A,5,5,6)
12 R-Mergesort(A[7..7])
13 L-Mergesort(A[7..7])
12 Merge(A,7,7,7)
8 Merge(A,5,6,7)
0 Merge(A,1,4,7)



Conclusion

Iterative algorithms :

- A function call causes the program to jump in a different place in the code, i.e. the first instruction of the called function
- Returning from a function call causes the program to return to the instruction next to the function call in the calling function

Recursive algorithms :

- A function call causes the program to jump to the first instruction of the currently executing function (the calling function)
- Returning from a function call causes the program to return to the instruction next to the function call in the same function

Function calls for iterative and recursive algorithms are implemented the same way using activation records and run time stacks.