Embedded Software Design Techniques

C programming 2: algorithm complexity, program execution model

Tsuyoshi Isshiki

Dept. Communications and Integrated Systems

Tokyo Institute of Technology

TAIST ICTES

Conduct by Asst. Prof. Dr. Kanjanapan Sukvichai

Lecture Outline

- Embedded software overview
  - What are “embedded systems” and “embedded software”?
- C programming 1: C language overview
  - Function, declaration, statement, expression
  - Data types, data structure, pointers and pointer dereferences
- C programming 2: algorithm complexity, program execution model
  - Bubble sort vs quick sort
  - Stack memory and program execution
- C programming 3: programming techniques in image processing
  - Dynamic memory allocation, image array implementation
  - Greyscaling, filtering, binarization, color quantization, dithering
- C programming 4: programming complex applications
  - Program development steps (ex. Huffman coding)
  - Binary tree construction, tree traversal
  - Bitstream handling
- Real time operating systems and application development
  - RTOS services, kernels
  - Context switching, task scheduling
  - Multi-task programming model

Understanding and Designing Algorithms

- In order to be able to write “good” programs, you need to be able to understand how algorithms work and how to design them
- “Sorting” is one of the basic exercise for algorithm design and programming where a variety of algorithms exists with a wide range of computational complexities
  - It is also commonly used function in any application programs
- We will learn how algorithm designs can be very critical in the program execution time
- Here, we will also learn how to manipulate arrays and pointers, and some programming techniques
- Then we will look closely to how programs manipulate data and implement function calls, how call arguments are passed to function parameters

“create_array”

```
#include <stdlib.h> /* required for srand() and rand() */
#include <stdio.h>
void swap(int * a0, int * a1)
{
      int t = *a0;
      *a0 = *a1;
      *a1 = t;
}
void create_array (int a[], int n, int rand_seed)
{
      int i;
      for(i = 0; i < n; i ++)
         a[i] = i;
      srand(rand_seed);
      for(i = 0; i < 2 * n; i ++){
         int j0 = rand() % n;
         int j1 = rand() % n;
         swap(&a[j0], &a[j1]);
      }
}
```

- `srand(rand_seed)`: initialize randomize seed
- `rand()` : get random integer (range: 0 to RAND_MAX = 0x7fff = 32767)
  - If srand() is not called before rand(), it is equivalent to calling srand(1)
- `create_array()` : randomly chooses a pair to swap inside the array
  - `rand() % n` ensures that `j0` and `j1` are within valid subscript range [0, n – 1]

“create_array”

```
void print_array2(int a[], int l, int r)
{
      int j;
      printf("a[%d:%d] = {", l, r);
      for(j = l; j <= r; j ++){
         if(j > l) printf(", ");
         printf("%d", a[j]);
      }
      printf("}\n");
}
#define N 10
#define RSEED 3
void main()
{
      int a[N];
      create array(a, N, RSEED);
                             1);
}
```

6 5 3 1 8 7 2 4

```
Output:
a[0:9] = {0, 5, 1, 3, 9, 7, 6, 8, 4, 2}
```

- `#define N 10` : string “N” is replaced with string “10” in the program source by the “preprocessor” before compilation
  - `int a[N];` is equivalent to `int a[10];`
  - `printArray2(a, 0, N - 1);` is equivalent to `printArray2(a, 0, 9);`
  - → C compiler precomputes 10 – 1 = 9 during code generation
- Using #define “preprocessor” is convenient when you want to change many constant values in the program simultaneously

“bubble_sort”

```
int comp_count = 0;
void bubble_sort(int a[], int n)
{
      int i, j, t = 0;
      for(i = 0; i < n; i ++){
         for(j = 1; j < n - i; j ++){
            if(a[j - 1] > a[j])
                  swap(&a[j - 1], &a[j]);
         }
         print_array2(a, 0, n - i - 1);
         comp_count += n - i - 1;
      }
}
```

```
Initial array:
a[0:9] = {0, 5, 1, 3, 9, 7, 6, 8, 4, 2}
i = 0   :
j = 1   : {0, 5, 1, 3, 9, 7, 6, 8, 4, 2}
j = 2   : {0, 1, 5, 3, 9, 7, 6, 8, 4, 2}
j = 3   : {0, 1, 3, 5, 9, 7, 6, 8, 4, 2}
j = 4   : {0, 1, 3, 5, 9, 7, 6, 8, 4, 2}
j = 5   : {0, 1, 3, 5, 7, 9, 6, 8, 4, 2}
j = 6   : {0, 1, 3, 5, 7, 6, 9, 8, 4, 2}
j = 7   : {0, 1, 3, 5, 7, 6, 8, 9, 4, 2}
j = 8   : {0, 1, 3, 5, 7, 6, 8, 4, 9, 2}
j = 9   : {0, 1, 3, 5, 7, 6, 8, 4, 2, 9}
```

- At the inner for-loop (`j = 1, 2, …, n - i - 1`)
  - `a[j - 1] <= a[j]` is ensured by `if(a[j - 1] > a[j]) swap2(&a[j - 1], &a[j]);`
- At the outer for-loop (`i = 0, 1, …, n - 1`)

he largest integer within `a[0]` to `a[n - i - 1]`

6 5 3 1 8 7 2 4

“bubble_sort”

```
int comp_count = 0;
void bubble_sort(int a[], int n)
{
      int i, j, t = 0;
      for(i = 0; i < n; i ++){
         for(j = 1; j < n – i; j ++){
            if(a[j - 1] > a[j])
                 swap(&a[j – 1], &a[j]);
         }
         print_array2(a, 0, n – i – 1);
         comp_count += n – i – 1;
      }
}
void main()
{
      int a[N];
      create_array(a, N, RSEED);
      print_array2(a, 0, N - 1);
      bubble_sort(a, N);
      print_array2(a, 0, N - 1);
}
```

- `int comp_count = 0;` : this is a global variable declaration (declared outside the function body)
  - Global variables are visible from inside the function body (after its declaration)
  - Initialization of global variables must be constant value (if no initialization is specified, it is initialized to 0 by default)
- `comp_count += n – i – 1;` : this accumulates the number of comparisons (`a[j - 1] > a[j]`) performed in the sorting function → a rough measure of computation time

“bubble_sort”

```
double comp_count = 0;
void bubble_sort(int a[], int n)
{
      int i, j, t = 0;
      for(i = 0; i < n; i ++){
         for(j = 1; j < n - i; j ++){
            if(a[j - 1] > a[j])
                  swap(&a[j - 1], &a[j]);
         }
         print_array2(a, 0, n - i - 1);
         comp_count += n - i - 1;
      }
}
```

```
Output:
a[0:9] = {0, 5, 1, 3, 9, 7, 6, 8, 4, 2}
a[0:9] = {0, 1, 3, 5, 7, 6, 8, 4, 2, 9}
a[0:8] = {0, 1, 3, 5, 6, 7, 4, 2, 8}
a[0:7] = {0, 1, 3, 5, 6, 4, 2, 7}
a[0:6] = {0, 1, 3, 5, 4, 2, 6}
a[0:5] = {0, 1, 3, 4, 2, 5}
a[0:4] = {0, 1, 3, 2, 4}
a[0:3] = {0, 1, 2, 3}
a[0:2] = {0, 1, 2}
a[0:1] = {0, 1}
a[0:9] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
comp_count = 45
```

- Total number of comparison is n * (n – 1) / 2 = O(n^2)
- Very simple form of sorting algorithm (but slow)

“quick_sort”

- Algorthm: Divide-and-Conquer strategy
  1. Select a “pivot” element in the array
  2. Partition the array so that
     - all elements smaller than the pivot element is to the left of the pivot
     - all elements greater than the pivot element is to the right of the pivot

       (elements that are equal to the pivot element can be at either side)

     → After this array partitioning, the pivot element will be correctly positioned with respect to the sorting order
  3. Apply the routines 1 and 2 on the sub-array left of the pivot element
  4. Apply the routines 1 and 2 on the sub-array right of the pivot element
- Ex:

  Initial array: `{0, 5, 1, 3, 9, 7, 6, 8, 4, 2}`

  → Pivot = 2 : `{0, 1}, 2, {3, 9, 7, 6, 8, 4, 5}`

  On sub-array: `{3, 9, 7, 6, 8, 4, 5}`

  → Pivot = 5 : `{3, 4}, 5, {6, 8, 9, 7}`

  On sub-array: `{6, 8, 9, 7}`

  → Pivot = 7 : `{6}, 7, {9, 8}`

  On sub-array: `{9, 8}`

  → Pivot = 8 : `{}, 8, {9}`

6 5 3 1 8 7 2 4

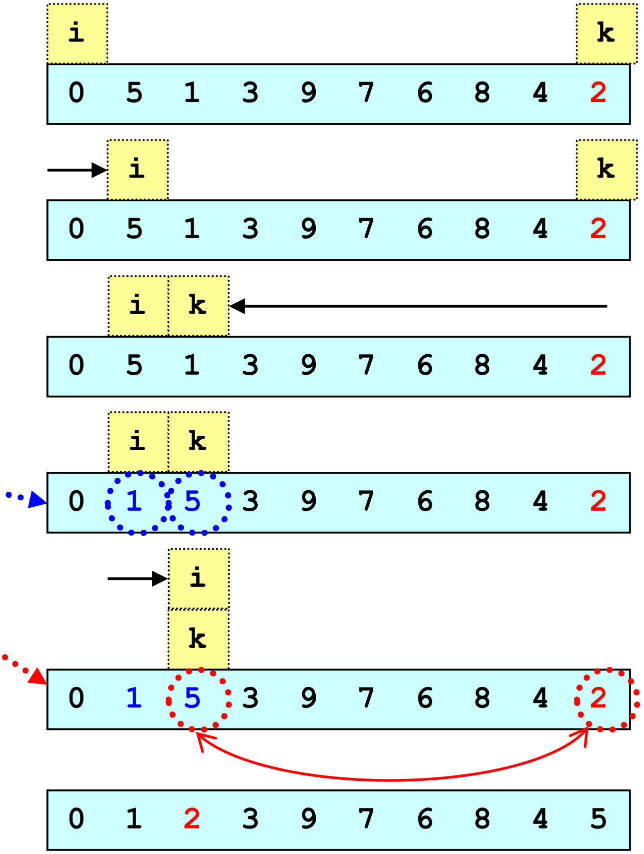
- Implementation issues
  - How to select a pivot element in the array?
    - Best pivot element choice is the median so that the array will be partitioned evenly, but finding the median element can be costly (above example simply chooses rightmost element as the pivot)
  - How to partition the array?

“quick_sort”

6 5 3 1 8 7 2 4

```
void print_pivot(int pivot, int a[], int l, int r)
{
        printf("pivot(%d): ", pivot);
        print_array2(a, l, r);
}
int partition(int a[], int left, int right)
{
        int pivot = a[right];
        int i = left, k = right;
        while(i < k){
            while(i < k && a[i] < pivot) i ++;
            while(i < k && a[k] >= pivot) k --;
            if(i < k) swap(&a[i], &a[k]);
        }
        if(right > k) swap(&a[right], &a[k]);
        print_pivot(pivot, a, left, right);
        comp_count += right - left - 1;
        return k;
}
```

pivot = a[9] = 2



i k

0 5 1 3 9 7 6 8 4 2

i k

0 5 1 3 9 7 6 8 4 2

i k

0 5 1 3 9 7 6 8 4 2

i k

0 1 5 3 9 7 6 8 4 2

i

k

0 1 5 3 9 7 6 8 4 2

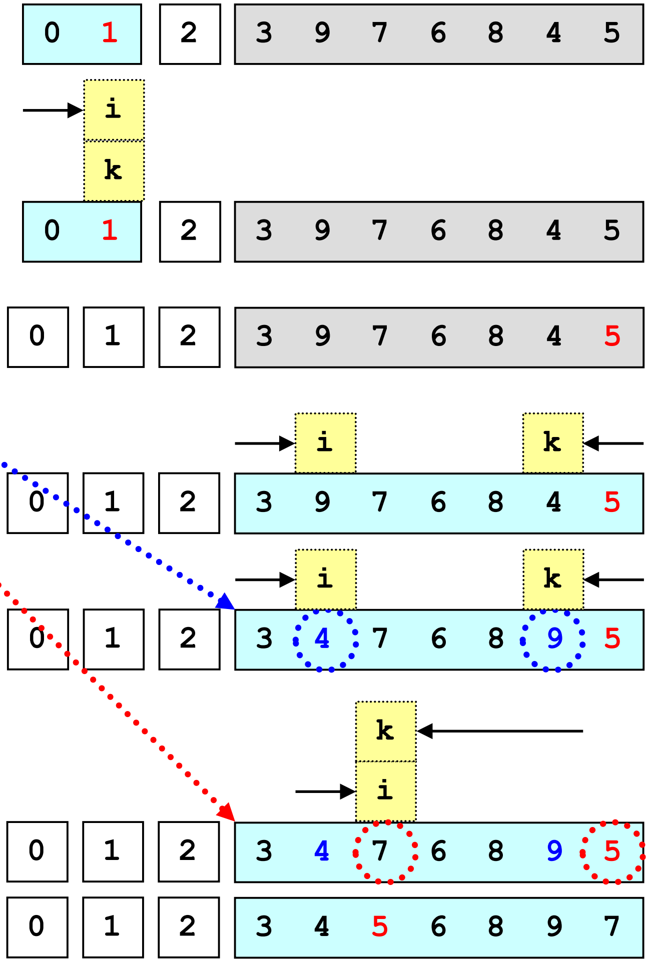
0 1 2 3 9 7 6 8 4 5

```
Output:
pivot(2): a[0:9] = {0, 1, 2, 3, 9, 7, 6, 8, 4, 5}
```

“quick_sort”

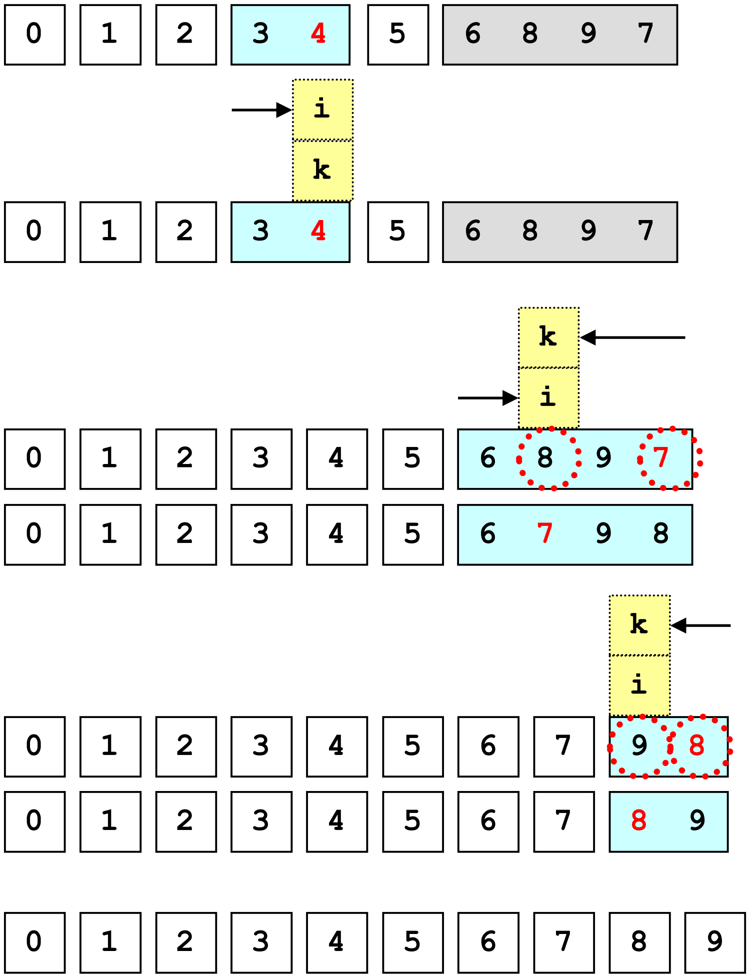
```
int partition(int a[], int left, int right)
{
        int pivot = a[right];
        int i = left, k = right;
        while(i < k){
            while(i < k && a[i] < pivot) i ++;
            while(i < k && a[k] >= pivot) k --;
            if(i < k) swap(&a[i], &a[k]);
        }
        if(right > k) swap(&a[right], &a[k]);
        print_pivot(pivot, a, left, right);
        comp_count += right - left - 1;
        return k;
}
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```

Recursive



“quick_sort”

```
int partition(int a[], int left, int right)
{
        int pivot = a[right];
        int i = left, k = right;
        while(i < k){
            while(i < k && a[i] < pivot) i ++;
            while(i < k && a[k] >= pivot) k --;
            if(i < k) swap(&a[i], &a[k]);
        }
        if(right > k) swap(&a[right], &a[k]);
        print_pivot(pivot, a, left, right);
        comp_count += right - left - 1;
        return k;
}
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```



“quick_sort”

```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
void main()
{
        int a[N];
        create_array(a, N, RSEED);
        print_array2(a, 0, N - 1);
        quick_sort(a, 0, N - 1);
        print_array2(a, 0, N - 1);
}
```

```
Output:
a[0:9] = {0, 5, 1, 3, 9, 7, 6, 8, 4, 2}
pivot(2): a[0:9] = {0, 1, 2, 3, 9, 7, 6, 8, 4, 5}
pivot(1): a[0:1] = {0, 1}
pivot(5): a[3:9] = {3, 4, 5, 6, 8, 9, 7}
pivot(4): a[3:4] = {3, 4}
pivot(7): a[6:9] = {6, 7, 9, 8}
pivot(8): a[8:9] = {8, 9}
a[0:9] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
comp_count = 15
```

- quick_sort() is initially called with left = 0, right = N – 1
- quick_sort() is then recursively called 2 times
- quick_sort is considered as the fastest sorting algorithm because:
  - It runs in O(n log n) on average (although worst case is O(n^2))
  - partition() function can be implemented efficiently requiring small number of swaps

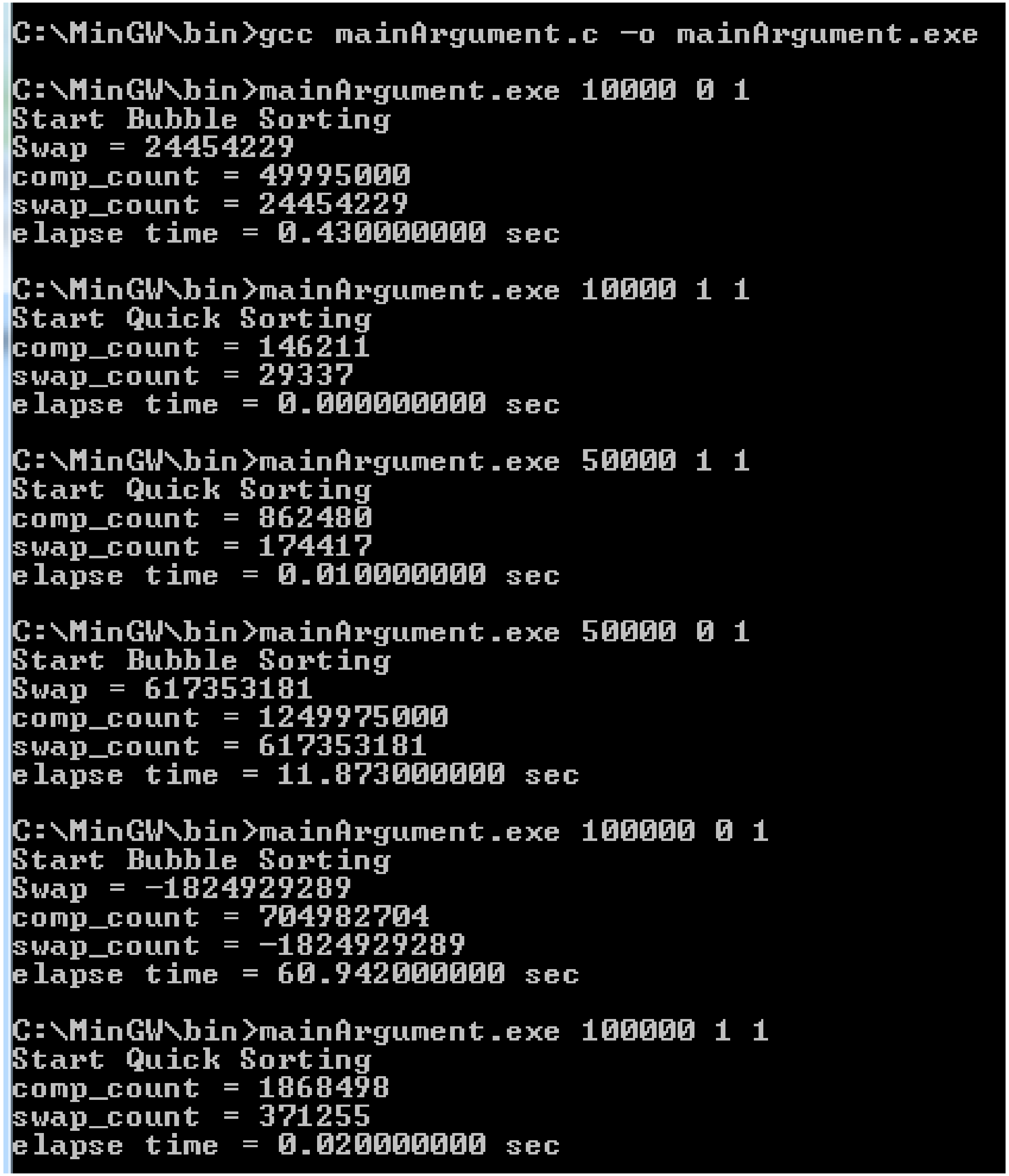
“bubble_sort” vs “quick_sort”

```
void create_array(int a[], int n, int rand_seed)
{
         int i;
         for(i = 0; i < n; i ++)
              a[i] = i;
         srand(rand_seed);
         for(i = 0; i < 2 * n; i ++){
              int j0 = ((rand() << 15) | rand()) % n;
              int j1 = ((rand() << 15) | rand()) % n;
              swap2(&a[j0], &a[j1]);
         }
}
```

- Need some modifications in create_array to test large arrays (WHY??)

array size	bubble_sort			quick_sort		
	# compares	# swaps	time (s)	# compares	# swaps	time (s)
10	45	27	0.000	14	7	0.000
100	4950	2623	0.000	558	145	0.000
1000	499500	238975	0.000	10024	2155	0.000
10000	49995000	24634442	0.640	142299	29760	0.000
100000	4999950000	2470083304	64.625	1929272	370026	0.015
1000000	499999500000	246693406350	6667.829	24850475	4426292	0.250

- Always be careful with the computational complexity when writing programs and designing algorithms
- Even $O(n^2)$ algorithms are not acceptable in some cases!!



```
C:\MinGW\bin>gcc mainArgument.c -o mainArgument.exe

C:\MinGW\bin>mainArgument.exe 10000 0 1
Start Bubble Sorting
Swap = 24454229
comp_count = 49995000
swap_count = 24454229
elapse time = 0.430000000 sec

C:\MinGW\bin>mainArgument.exe 10000 1 1
Start Quick Sorting
comp_count = 146211
swap_count = 29337
elapse time = 0.000000000 sec

C:\MinGW\bin>mainArgument.exe 50000 1 1
Start Quick Sorting
comp_count = 862480
swap_count = 174417
elapse time = 0.010000000 sec

C:\MinGW\bin>mainArgument.exe 50000 0 1
Start Bubble Sorting
Swap = 617353181
comp_count = 1249975000
swap_count = 617353181
elapse time = 11.873000000 sec

C:\MinGW\bin>mainArgument.exe 100000 0 1
Start Bubble Sorting
Swap = -1824929289
comp_count = 704982704
swap_count = -1824929289
elapse time = 60.942000000 sec

C:\MinGW\bin>mainArgument.exe 100000 1 1
Start Quick Sorting
comp_count = 1868498
swap_count = 371255
elapse time = 0.020000000 sec
```

Based on
i5-8 generation
16 GB DDR4
Windows 10

“main” function arguments

- “main” function prototypes
  - `void main()`
  - `int main()`
  - `void main(int argc, char * argv[])`
  - `int main(int argc, char * argv[])`
    - `int argc` : # of command arguments
    - `char * argv[]` : array of command strings
    - Use command arguments to pass parameters to main() such as array size, sort algorithm type, random seed number.
  - Return value of “main” function: used to signal the program execution status to the OS
- Ex:
  - `C:\mywork> a.exe 300 0 1`
  - argc = 4
  - argv[0] = “a.exe” (executable file name created by C compiler)
  - argv[1] = “300” (array size)
  - argv[2] = “0” (sort algorithm type)
  - argv[3] = “1” (random seed number)

Parsing “main” command arguments

```
#define N 1000000  /* this is the maximum array size */
#include <time.h> /* for using clock() and CLOCKS_PER_SEC */
int a[N]; /* declare array a as global because the array size is HUGE! */
int main(int argc, char * argv[])
{
       int n, rand_seed = 1, sort_type = 1, clk;
       if(argc < 2 || argc > 4){ /* # of command arguments must be 2, 3 or 4 */
          printf("Invalid command!\n");
          printf("usage: %s <array_size> <rand_seed> <sort-type>\n", argv[0]);
          printf("<rand_seed>, <sort-type> : optional\n");
          printf("sort-type = 0:bubble-sort, 1:quick-sort\n");
          return 0;
       }
       sscanf(argv[1], "%d", &n); /* read array size n */
       if(n <= 0 || n > N){ /* check that array size n is valid */
          printf("Invalid array size!\n");
          printf("Array size must be between %d and %d\n", 1, N);
          return 0;
       }
       if(argc >= 3){ /* read sort_type if 3rd argument is present */
          sscanf(argv[2], "%d", &sort_type);
       }
       if(argc >= 4){ /* read rand_seed if 4th argument is present */
          sscanf(argv[3], "%d", &rand_seed);
       }

/* main() continues to next page ... */
```

Parsing "main" command arguments

```
/* ... main() continued from previous page */

        if(sort_type < 0 || sort_type > 1){
            printf("Invalid sort type!\n");
            printf("sort-type = 0:bubble-sort, 1:quick-sort\n");
            return 0;
        }
        create_array(a, n, rand_seed); /* use n and rand_seed from command argument */
        printArray2(a, 0, n - 1);
        clk = clock(); /* get the current clock value : start timer from here */
        switch(sort_type){
            case 0: /* call bubble_sort */
                        bubble_sort(a, n);
                        break;
            case 1: /* call quick_sort */
                        quick_sort(a, 0, n - 1);
                        break;
        }
        clk = clock() - clk; /* get the current clock value : clk is the elapse time */
        printArray2(a, 0, n - 1);
        printf("comp_count = %.f\nswap_count = %.f\n", comp_count, swap_count);
        /* CLOCKS_PER_SEC : # of clock ticks per second (usually 1000 ticks/sec) */
        printf("elapse time = %.3f sec\n", (double) clk / (double) CLOCKS_PER_SEC);
        return 0;
}
```

- How to count # of swaps (swap_count)?
- comp_count and swap_count modified in the program as "double" type. Why?

Controlling “printf” Messages During Large Program Benchmarking

- Don’t want to display arrays with thousands of elements
- But still want to check that the program is working …

```
void print_array3(int a[], int l, int r) /* call this before and after the sort */
{
       int j;
       printf("a[%d:%d] = {", l, r);
       if(r - l < 20){ /* if the length is 20 or length, then print the whole array */
          for(j = l; j <= r; j ++){
                    if(j > l) printf(", ");
                    printf("%d", a[j]);
          }
       }
       else{  /* otherwise, print the first 5 elements and last 5 elements */
          for(j = l; j < l + 5; j ++){
                    if(j > l) printf(", ");
                    printf("%d", a[j]);
          }
          printf(", ... ");
          for(j = r - 4; j <= r; j ++)
                    printf(", %d", a[j]);
       }
       printf("}\n");
}
```

```
Output:
a[0:99999] = {3530, 95600, 33206, 98141, 57507, ... , 27894, 4331, 29430, 97238, 3462}
a[0:99999] = {0, 1, 2, 3, 4, ... , 99995, 99996, 99997, 99998, 99999}
```

Controlling “printf” Messages During Large Program Benchmarking

- For intermediate sorting results, simply skip printf calls.

1. “Comment out” printArray2() calls

```
    /* print_array2(a, left, right); */
```

2. Use the “preprocessor” `#if ... #endif`

```
    #ifdef ENABLE_PRINT_ARRAY2
    print_array2(a, left, right);
    #endif
```

→ Above methods can be time consuming if there are many places in the code calling the print functions

3. Use `#define` to replace the function call with a “null” string:

```
    #ifdef ENABLE_PRINT_ARRAY2
    /* put the actual function definition here */
    void print_array2(int a[], int l, int r){ ... }
    void print_pivot(int pivot){ ... }
    #else
    #define print_array2(a, l, r)
    #define print_pivot(pivot)
    /* this is a preprocessor “macro” with empty definition body */
    /* so, “print_array2(a, left, right);” will be replaced with “;” */
    #endif
```

→ This technique is very common in various debugging scenarios (such as ASSERT())

Understanding How Programs Use Memory to Manipulate Data

- Next, we will look into details of how the program behaves
  - How is function calls implemented?
    - How are the call arguments passed to the function parameter variables?
    - How is the data at the caller's function restored after the call returns?
    - How can a function called recursively?
  - Where are the variables stored in memory?
    - Global variables
    - Parameter variables
    - Local variables

Variable Lifetime and Memory Allocation

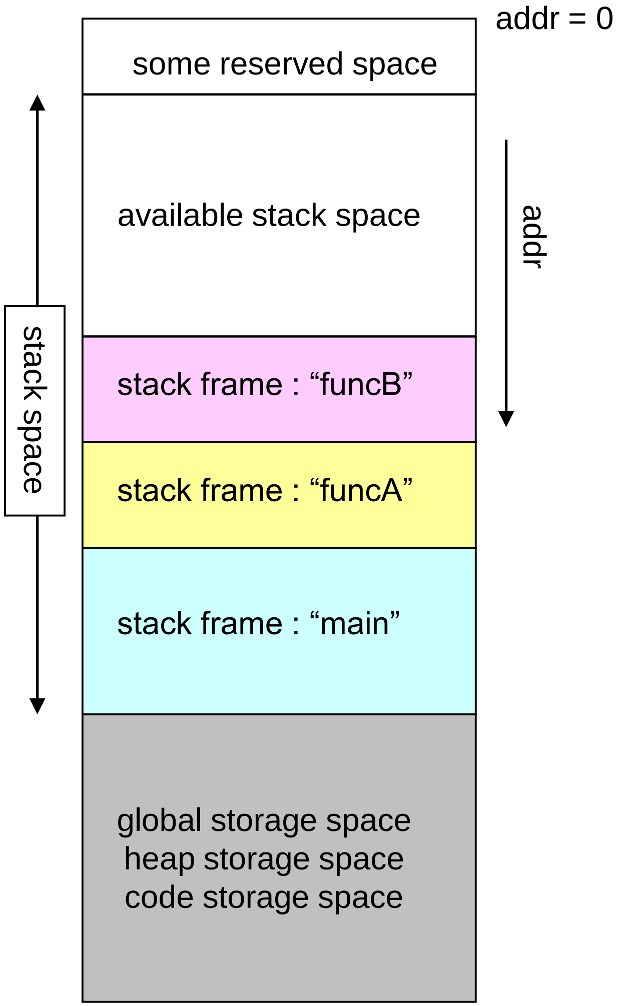
- Global lifetime storage variables:
  - Absolute address is determined at compile time
  - Variable storage exists during the entire program execution
- Local lifetime storage variables:
  - Relative address (against the “stack pointer”) is determined at compile time
  - Variable storage exists only during the execution of the compound-statement it is included in

```
{
   int a = ...;
   /* a is alive only inside this compound-statement */
}
/* variable “a” cannot be accessed here */
```

  - Function parameter variables are also local lifetime storage variables (they are assumed to be included in the top-level compound-statement of the function body)

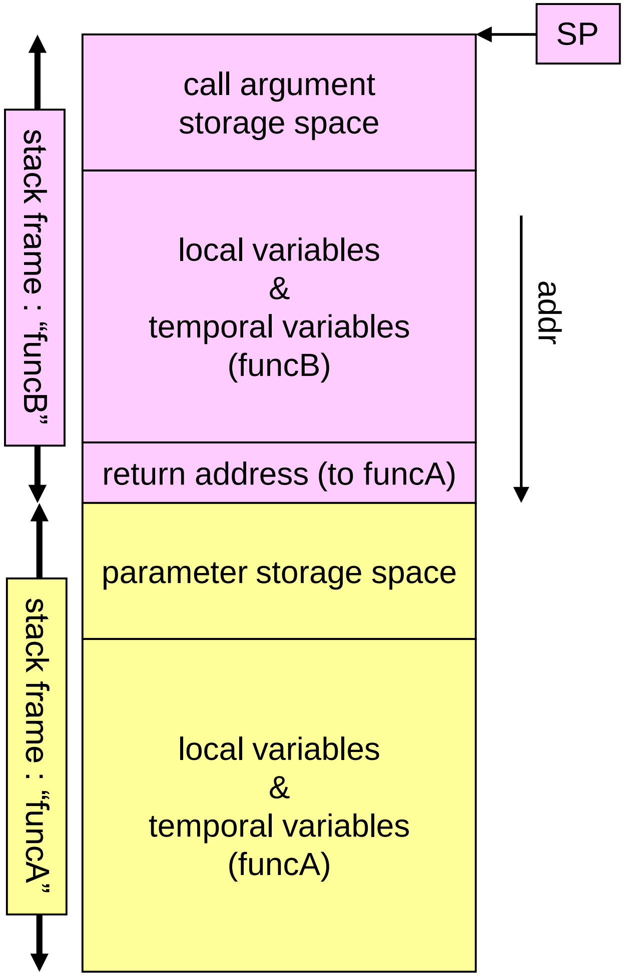
Stack

- Stack is a block of memory where the following objects are stored
  - Local lifetime storage variables (including function parameter variables)
  - Temporal storage for expression evaluations
  - Function return value
  - Function return address (program counter after the function call instruction)
- Stack space is separated from global storage space, heap storage space (for dynamically allocated variables) and code space (which stores execution codes)
  - String literals are usually stored in the global storage space
- Stack space size is usually not so large (typically 64K bytes in PC environment), so large arrays should not be declared locally
- Stack frame
  - Stack frame is created whenever a function is called, and removed when the function returns
  - Stack frame reserves the memory space in the stack for storing local variables
  - Instantiation of stack frames on each function call makes it possible to implement recursive function calls



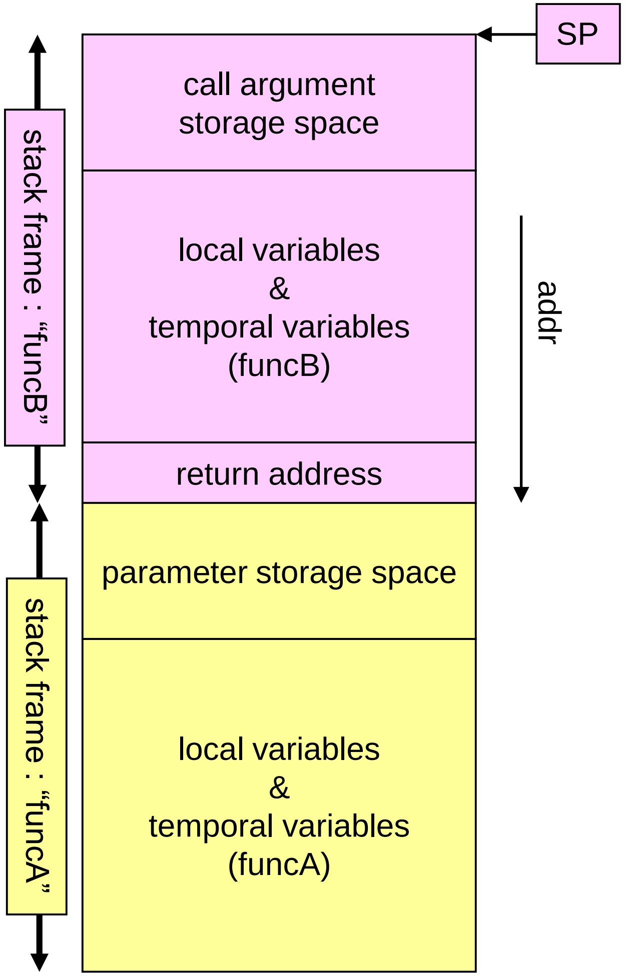
Stack

- SP (stack pointer) : base address register for accessing local variables
  - SP is set at the beginning of the current stack frame
  - Address of local variable (including parameter variable) is calculated by adding the relative address (determined by the compiler) to SP
- Stack frame is usually organized in the following order
  - Call argument storage space: function call arguments are copied to this region which will be accessible by the callee function (function that is called). The size of this storage space is set as the maximum required space among the different function calls made inside the current function
  - Local storage space: local variables and temporal variables are stored in this region.
  - Return address storage space: a single word space for storing the return address (back to the caller function)
  - Parameter storage space: this overlaps with the caller's "call argument storage space" which is located below the current stack frame. Function return value is usually stored in this region (so that the caller can retrieve the return value)


SP
call argument storage space
local variables & temporal variables (funcB)
return address (to funcA)
parameter storage space
local variables & temporal variables (funcA)
stack frame : "funcB"
stack frame : "funcA"
addr

Stack

- Stack model in the *following slides* (for the sake of simplicity)
  - All parameters are copied to stack
  - All local variables are stored in stack
  - No temporal variables are assumed.
  - Stack space starts from address 0x10000 towards 0x00000
  - → These assumptions usually do not apply to real machines
    - Registers are often used to pass parameters
    - Local scalar variables are often stored in registers instead of stack
    - Register contents need to be stored in the stack during function calls
  - Storing and retrieving "return address" is explained differently than the commonly used PUSH & POP operations
- Notation
  - MEM[SP + 12] → memory word at address (SP + 12)

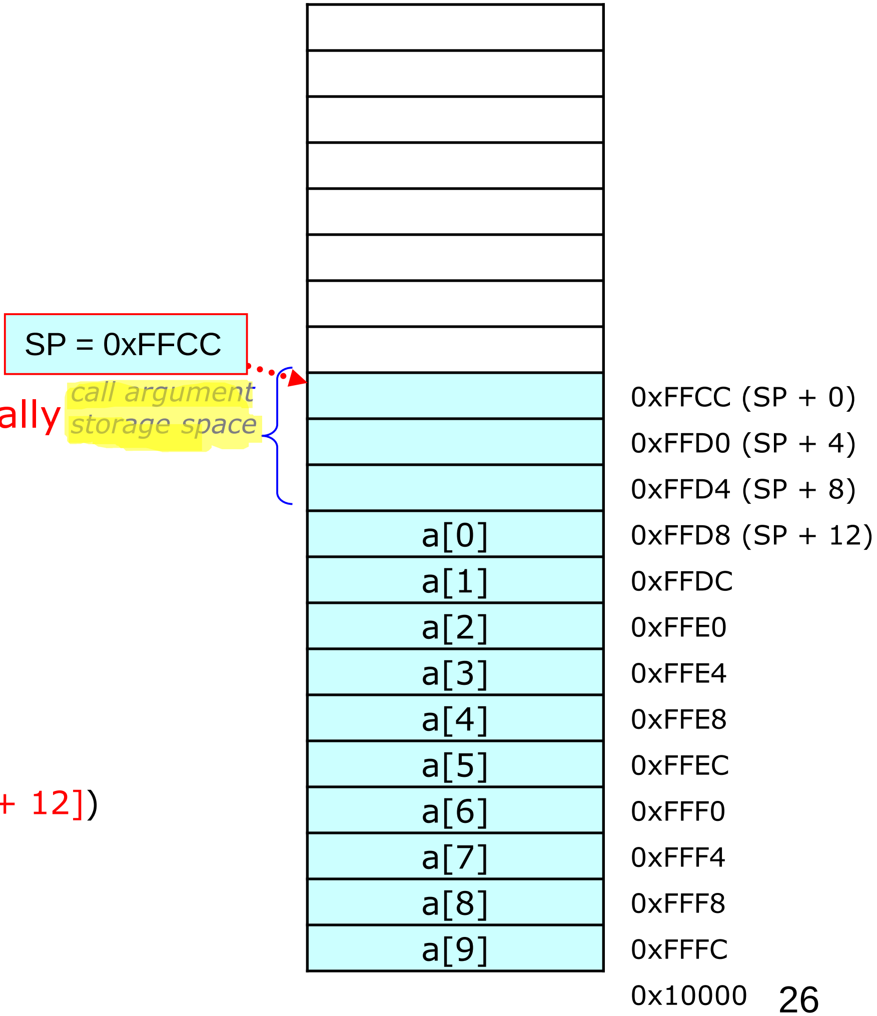

SP
call argument storage space
local variables & temporal variables (funcB)
return address
parameter storage space
local variables & temporal variables (funcA)
stack frame : "funcB"
stack frame : "funcA"
addr

Stack Frame “main”

```
#define N 10
#define RSEED 3
void main()
{
        int a[N]; /* size = 4N = 40 bytes */
        create_array(a, N, RSEED);
        print_array2(a, 0, N - 1);
        quick_sort(a, 0, N – 1);
        print_array2(a, 0, N - 1);
}
```

- In this example, array “a” is declared locally inside the “main”

- Before entering “main”: SP = 0x10000
- Enter “main”:
  - SP = SP – 52 = 0xFFCC
  - 12 bytes for storing 3 argument values for calling `create_array`, `print_array2`, `quick_sort`
  - 40 bytes for storing “int a[10]”
    (head element of “a” is stored at MEM[SP + 12])
    (a = SP + 12)



SP = 0xFFCC

call argument storage space

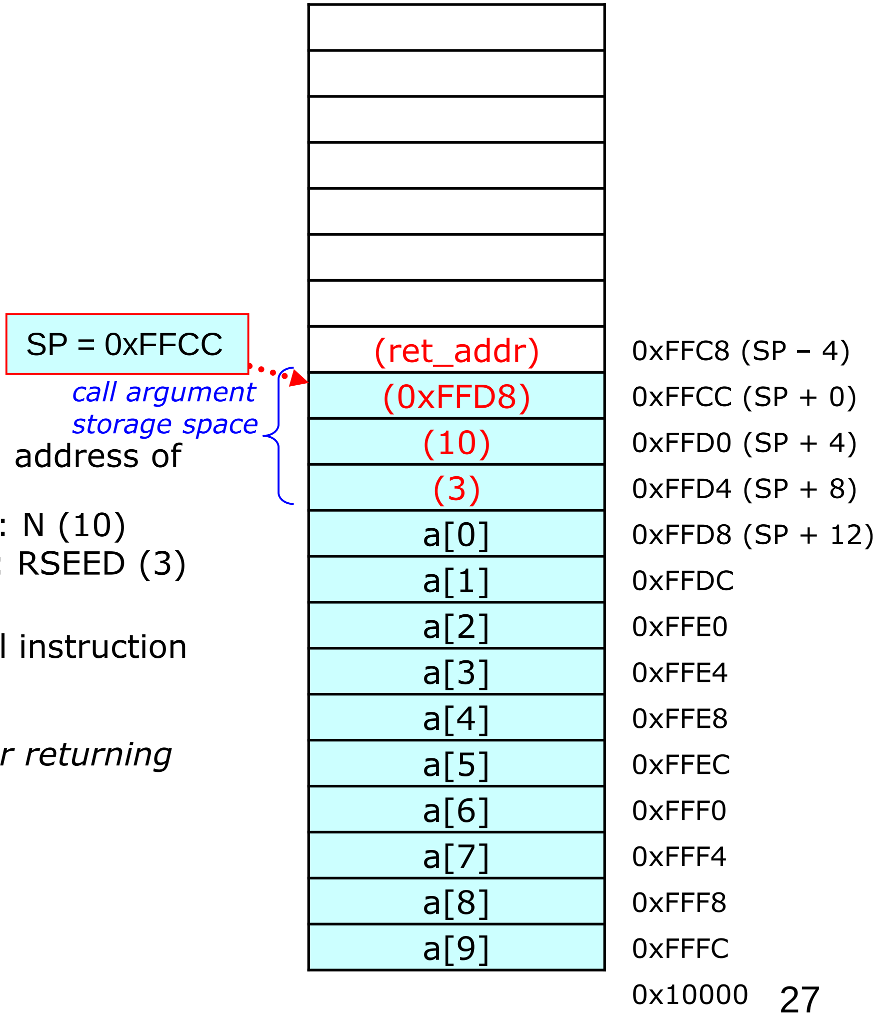
	0xFFCC (SP + 0)
	0xFFD0 (SP + 4)
	0xFFD4 (SP + 8)
a[0]	0xFFD8 (SP + 12)
a[1]	0xFFDC
a[2]	0xFFE0
a[3]	0xFFE4
a[4]	0xFFE8
a[5]	0xFFEC
a[6]	0xFFF0
a[7]	0xFFF4
a[8]	0xFFF8
a[9]	0xFFFC
	0x10000

Stack Frame "main"

```
#define N 10
#define RSEED 3
void main()
{
       int a[N]; /* size = 4N = 40 bytes */
       create_array(a, N, RSEED);
       print_array2(a, 0, N - 1);
       quick_sort(a, 0, N – 1);
       print_array2(a, 0, N - 1);
}
```

- Before calling "create_array":
  - Copy 1st argument value to MEM[SP + 0] : address of "a" (SP + 12 = 0xFFD8)
  - Copy 2nd argument value to MEM[SP + 4] : N (10)
  - Copy 3rd argument value to MEM[SP + 8] : RSEED (3)
- Call "create_array":
  - Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - Jump to "create_array"
  - ← *Program resumes at I_MEM[ret_addr] after returning from "create_array"*

    (I_MEM : instruction memory)



SP = 0xFFCC

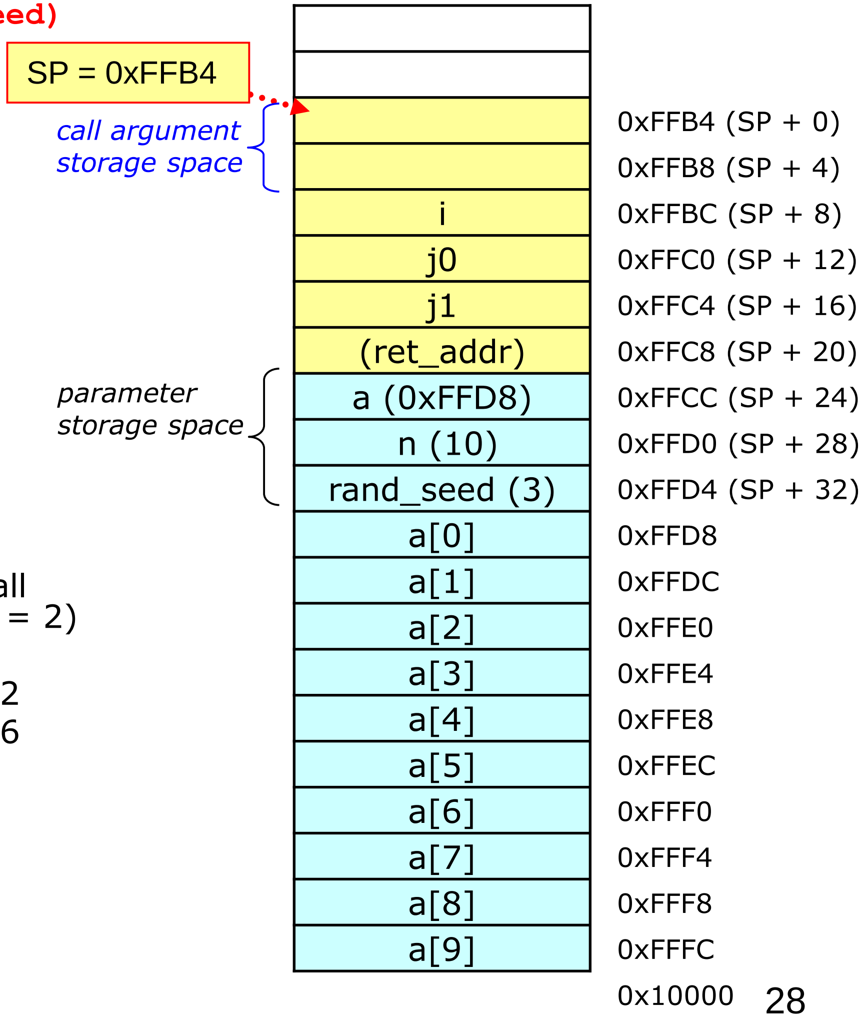
call argument storage space

(ret_addr)	0xFFC8 (SP – 4)
(0xFFD8)	0xFFCC (SP + 0)
(10)	0xFFD0 (SP + 4)
(3)	0xFFD4 (SP + 8)
a[0]	0xFFD8 (SP + 12)
a[1]	0xFFDC
a[2]	0xFFE0
a[3]	0xFFE4
a[4]	0xFFE8
a[5]	0xFFEC
a[6]	0xFFF0
a[7]	0xFFF4
a[8]	0xFFF8
a[9]	0xFFFC
	0x10000

Stack Frame “create_array”

```
void create_array (int a[], int n, int rand_seed)
{
        int i;
        for(i = 0; i < n; i ++)
            a[i] = i;
        srand(rand_seed);
        for(i = 0; i < 2 * n; i ++){
            int j0 = rand() % n;
            int j1 = rand() % n;
            swap(&a[j0], &a[j1]);
        }
}
```

- Enter “create_array”:
  - SP = SP – 24 = 0xFFB4
  - 8 bytes for storing 2 argument values for call swap2 (max # of arguments for these calls = 2)
  - 4 bytes for storing “int i” : &(i) = SP + 8
  - 4 bytes for storing “int j0” : &(j0) = SP + 12
  - 4 bytes for storing “int j1” : &(j1) = SP + 16
  - 4 bytes for storing ret_addr to the caller
- Parameter address
  - SP + 24 : int a[] (address of “a”)
  - SP + 28 : int n
  - SP + 32 : int rand_seed



SP = 0xFFB4

call argument storage space

parameter storage space

	0xFFB4 (SP + 0)
	0xFFB8 (SP + 4)
i	0xFFBC (SP + 8)
j0	0xFFC0 (SP + 12)
j1	0xFFC4 (SP + 16)
(ret_addr)	0xFFC8 (SP + 20)
a (0xFFD8)	0xFFCC (SP + 24)
n (10)	0xFFD0 (SP + 28)
rand_seed (3)	0xFFD4 (SP + 32)
a[0]	0xFFD8
a[1]	0xFFDC
a[2]	0xFFE0
a[3]	0xFFE4
a[4]	0xFFE8
a[5]	0xFFEC
a[6]	0xFFF0
a[7]	0xFFF4
a[8]	0xFFF8
a[9]	0xFFFC
	0x10000

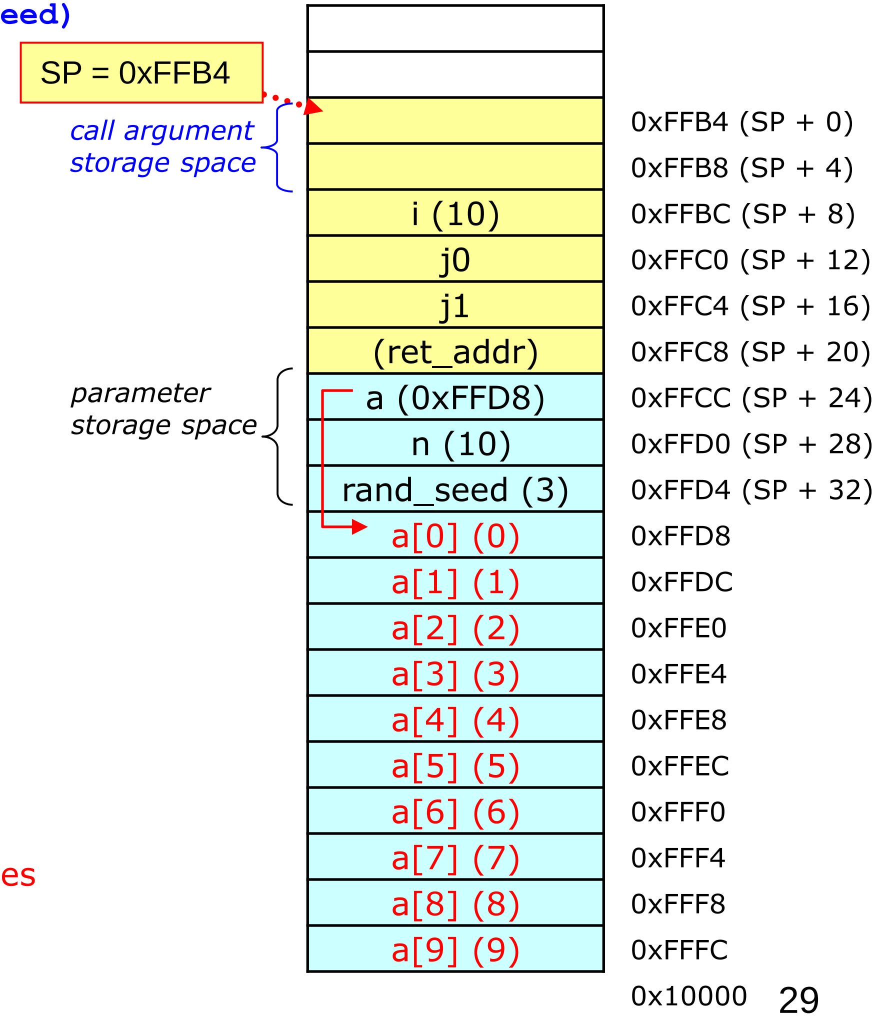
Stack Frame “create_array”

```
void create_array (int a[], int n, int rand_seed)
{
        int i;
        for(i = 0; i < n; i ++)
            a[i] = i;
        srand(rand_seed);
        for(i = 0; i < 2 * n; i ++){
            int j0 = rand() % n;
            int j1 = rand() % n;
            swap(&a[j0], &a[j1]);
        }
}
```

```
for(i = 0; i < n; i ++)
            a[i] = i;
```

write 0 to MEM[0xFFD8 + 0]
write 1 to MEM[0xFFD8 + 4]
write 2 to MEM[0xFFD8 + 8]
… … …

address of “a”   1 element = 4 bytes



SP = 0xFFB4

call argument storage space

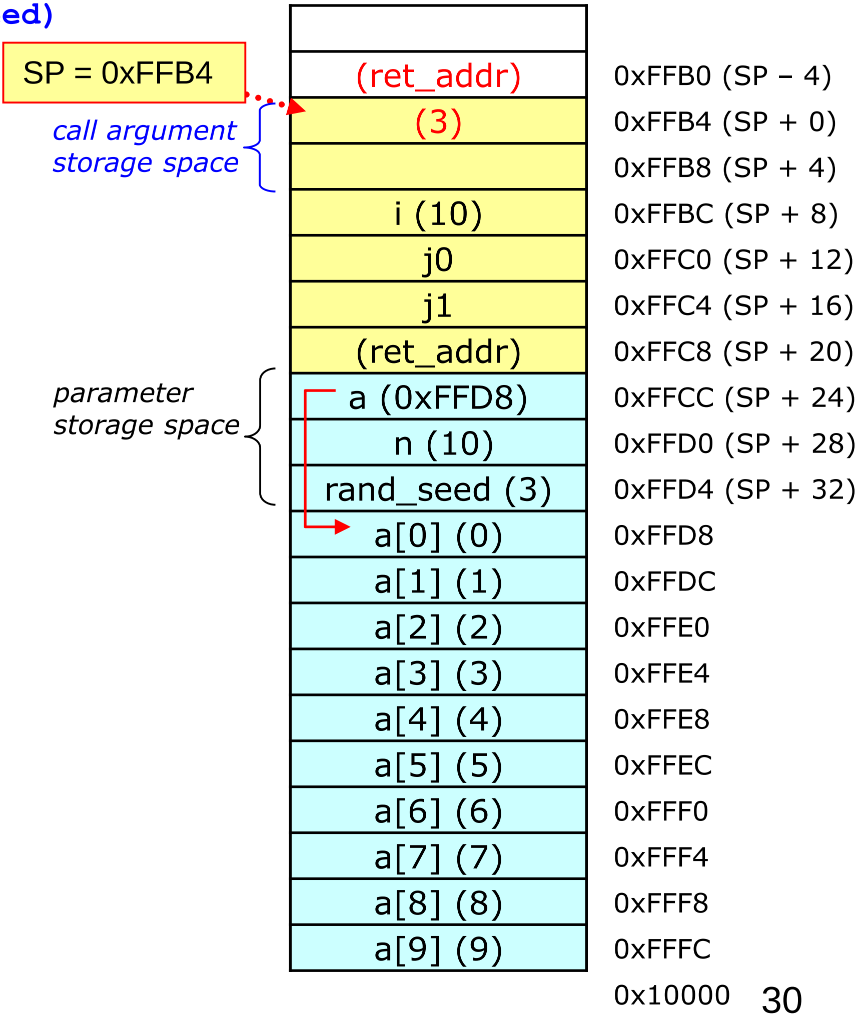
parameter storage space

	0xFFB4 (SP + 0)
	0xFFB8 (SP + 4)
i (10)	0xFFBC (SP + 8)
j0	0xFFC0 (SP + 12)
j1	0xFFC4 (SP + 16)
(ret_addr)	0xFFC8 (SP + 20)
a (0xFFD8)	0xFFCC (SP + 24)
n (10)	0xFFD0 (SP + 28)
rand_seed (3)	0xFFD4 (SP + 32)
a[0] (0)	0xFFD8
a[1] (1)	0xFFDC
a[2] (2)	0xFFE0
a[3] (3)	0xFFE4
a[4] (4)	0xFFE8
a[5] (5)	0xFFEC
a[6] (6)	0xFFF0
a[7] (7)	0xFFF4
a[8] (8)	0xFFF8
a[9] (9)	0xFFFC
	0x10000

Stack Frame “create_array”

```
void create_array (int a[], int n, int rand_seed)
{
        int i;
        for(i = 0; i < n; i ++)
            a[i] = i;
        srand(rand_seed);
        for(i = 0; i < 2 * n; i ++){
            int j0 = rand() % n;
            int j1 = rand() % n;
            swap(&a[j0], &a[j1]);
        }
}
```

- Before calling “srand”:
  - Copy 1st argument value to MEM[SP + 0]: rand_seed (3)
- Call “srand”:
  - Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - Jump to “srand”
  - ← *Program resumes at I_MEM[ret_addr] after returning from “srand”*



SP = 0xFFB4

call argument storage space

parameter storage space

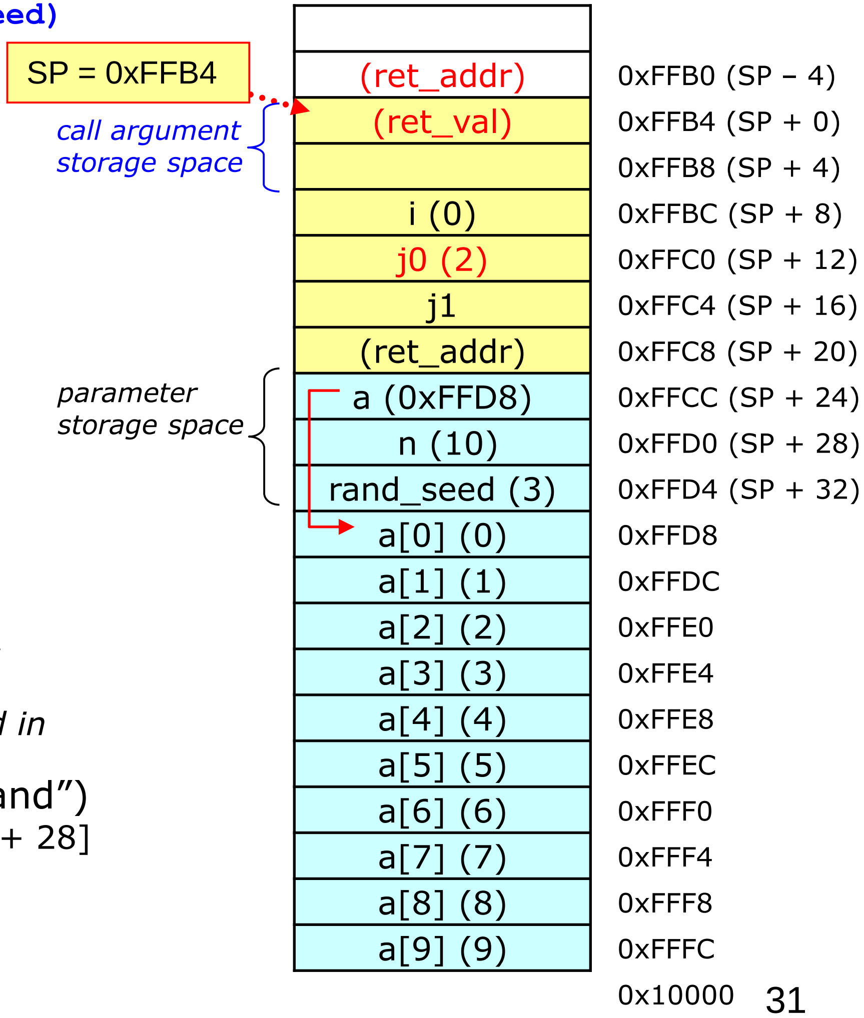
(ret_addr)	0xFFB0 (SP – 4)
(3)	0xFFB4 (SP + 0)
	0xFFB8 (SP + 4)
i (10)	0xFFBC (SP + 8)
j0	0xFFC0 (SP + 12)
j1	0xFFC4 (SP + 16)
(ret_addr)	0xFFC8 (SP + 20)
a (0xFFD8)	0xFFCC (SP + 24)
n (10)	0xFFD0 (SP + 28)
rand_seed (3)	0xFFD4 (SP + 32)
a[0] (0)	0xFFD8
a[1] (1)	0xFFDC
a[2] (2)	0xFFE0
a[3] (3)	0xFFE4
a[4] (4)	0xFFE8
a[5] (5)	0xFFEC
a[6] (6)	0xFFF0
a[7] (7)	0xFFF4
a[8] (8)	0xFFF8
a[9] (9)	0xFFFC
	0x10000

Stack Frame “create_array”

```
void create_array (int a[], int n, int rand_seed)
{
        int i;
        for(i = 0; i < n; i ++)
            a[i] = i;
        srand(rand_seed);
        for(i = 0; i < 2 * n; i ++){
            int j0 = rand() % n;
            int j1 = rand() % n;
            swap(&a[j0], &a[j1]);
        }
}
```

- Call 1st “rand”: (no arguments)
  - Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - Jump to “rand”
  - ← *Program resumes at I_MEM[ret_addr] after returning from “rand”*
  - ← *Return value (ret_val) from “rand” is stored in MEM[SP + 0]*
- **j0 = rand() % n** : (after return from “rand”)
  - MEM[SP + 12] ← MEM[SP + 0] % MEM[SP + 28]

(here, assume j0 = 2)



SP = 0xFFB4

call argument storage space

parameter storage space

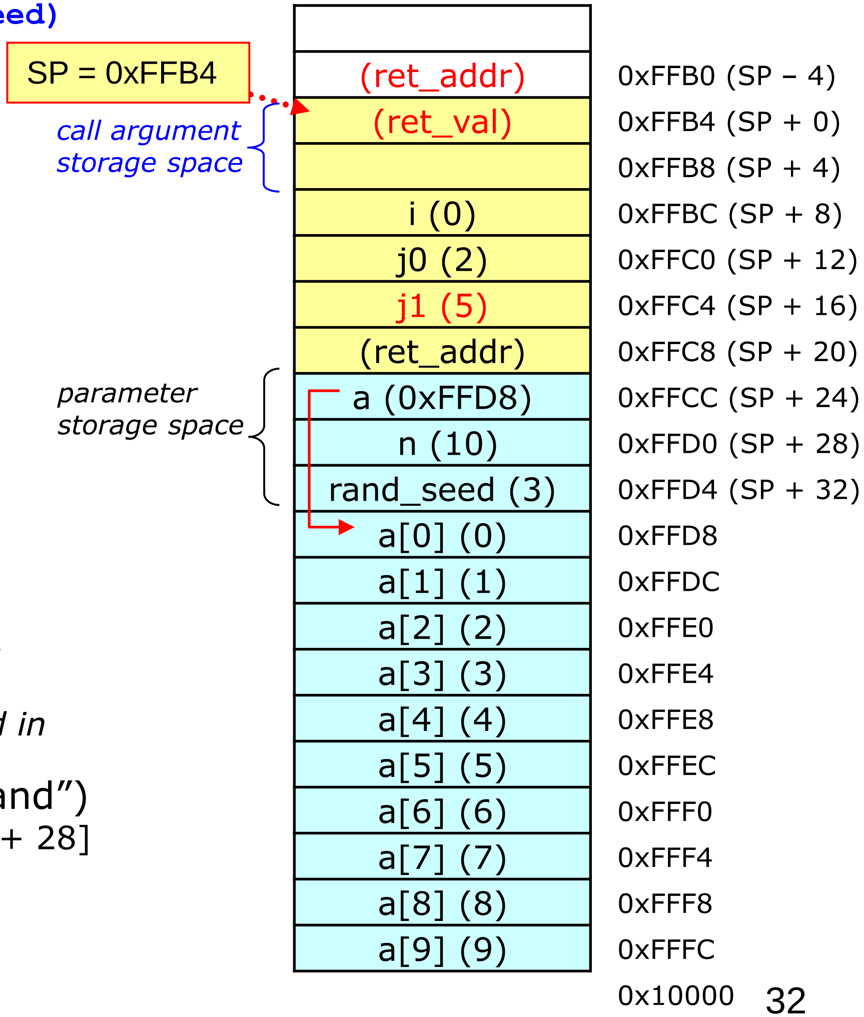
(ret_addr)	0xFFB0 (SP – 4)
(ret_val)	0xFFB4 (SP + 0)
	0xFFB8 (SP + 4)
i (0)	0xFFBC (SP + 8)
j0 (2)	0xFFC0 (SP + 12)
j1	0xFFC4 (SP + 16)
(ret_addr)	0xFFC8 (SP + 20)
a (0xFFD8)	0xFFCC (SP + 24)
n (10)	0xFFD0 (SP + 28)
rand_seed (3)	0xFFD4 (SP + 32)
a[0] (0)	0xFFD8
a[1] (1)	0xFFDC
a[2] (2)	0xFFE0
a[3] (3)	0xFFE4
a[4] (4)	0xFFE8
a[5] (5)	0xFFEC
a[6] (6)	0xFFF0
a[7] (7)	0xFFF4
a[8] (8)	0xFFF8
a[9] (9)	0xFFFC
	0x10000

Stack Frame “create_array”

```
void create_array (int a[], int n, int rand_seed)
{
        int i;
        for(i = 0; i < n; i ++)
            a[i] = i;
        srand(rand_seed);
        for(i = 0; i < 2 * n; i ++){
            int j0 = rand() % n;
            int j1 = rand() % n;
            swap(&a[j0], &a[j1]);
        }
}
```

- Call 2nd “rand”: (no arguments)
  - – Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - – Jump to “rand”
  - ← *Program resumes at I_MEM[ret_addr] after returning from “rand”*
  - ← *Return value (ret_val) from “rand” is stored in MEM[SP + 0]*
- **j1 = rand() % n** : (after return from “rand”)
  - – MEM[SP + 16] ← MEM[SP + 0] % MEM[SP + 28]

(here, assume j1 = 5)



SP = 0xFFB4

call argument storage space

parameter storage space

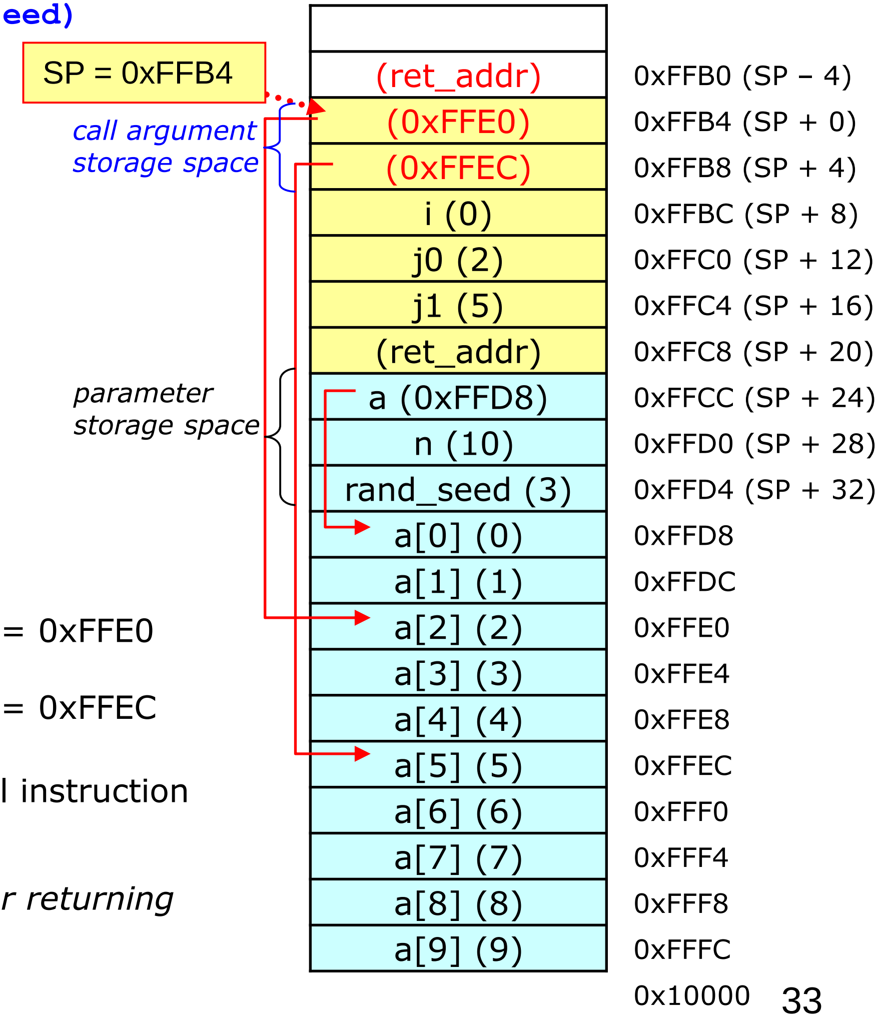
(ret_addr)	0xFFB0 (SP – 4)
(ret_val)	0xFFB4 (SP + 0)
	0xFFB8 (SP + 4)
i (0)	0xFFBC (SP + 8)
j0 (2)	0xFFC0 (SP + 12)
j1 (5)	0xFFC4 (SP + 16)
(ret_addr)	0xFFC8 (SP + 20)
a (0xFFD8)	0xFFCC (SP + 24)
n (10)	0xFFD0 (SP + 28)
rand_seed (3)	0xFFD4 (SP + 32)
a[0] (0)	0xFFD8
a[1] (1)	0xFFDC
a[2] (2)	0xFFE0
a[3] (3)	0xFFE4
a[4] (4)	0xFFE8
a[5] (5)	0xFFEC
a[6] (6)	0xFFF0
a[7] (7)	0xFFF4
a[8] (8)	0xFFF8
a[9] (9)	0xFFFC
	0x10000

Stack Frame “create_array”

```
void create_array (int a[], int n, int rand_seed)
{
        int i;
        for(i = 0; i < n; i ++)
            a[i] = i;
        srand(rand_seed);
        for(i = 0; i < 2 * n; i ++){
            int j0 = rand() % n;
            int j1 = rand() % n;
            swap(&a[j0], &a[j1]);
        }
}
```

- Before calling “swap”:
  (here, assume j0 = 2, j1 = 5)
  - Copy 1st argument value to MEM[SP + 0]:
    &a[j0] = (a) + (j0 * 4) = 0xFFD8 + 2 * 4 = 0xFFE0
  - Copy 2nd argument value to MEM[SP + 4]:
    &a[j1] = (a) + (j1 * 4) = 0xFFD8 + 5 * 4 = 0xFFEC
- Call “swap”:
  - Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - Jump to “swap2”

  ← *Program resumes at I_MEM[ret_addr] after returning from “swap”*



SP = 0xFFB4

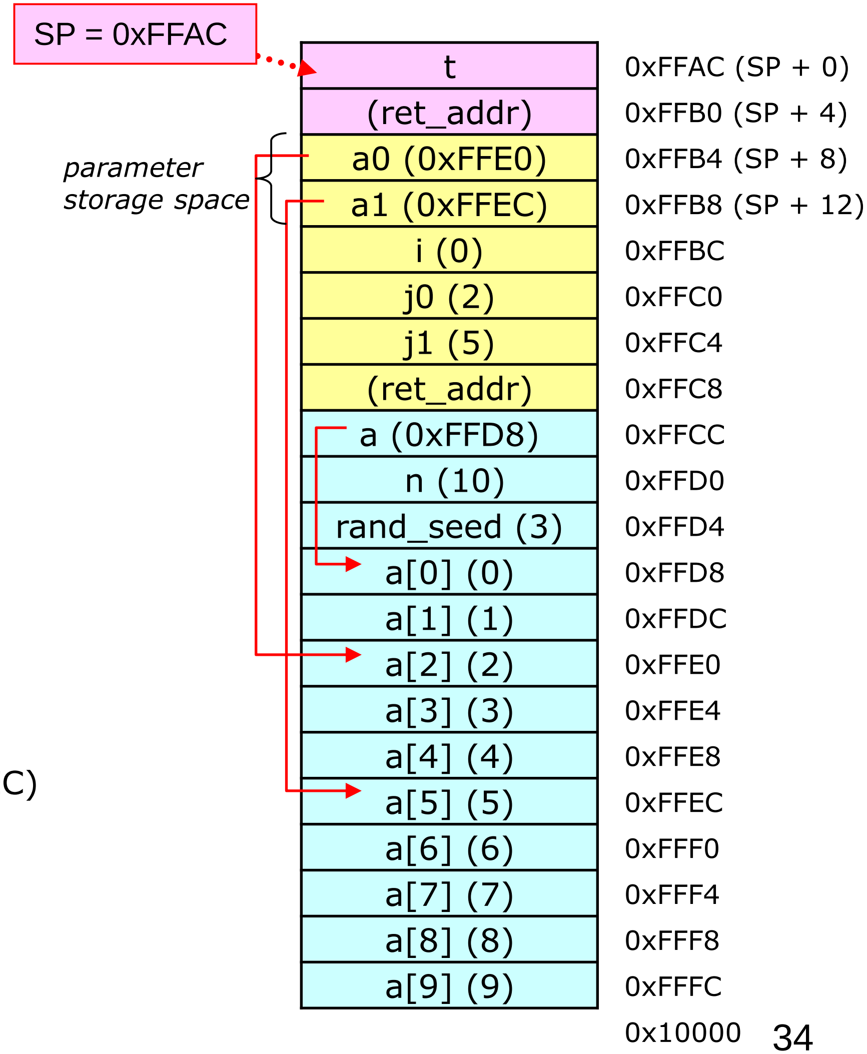
call argument storage space

parameter storage space

(ret_addr)	0xFFB0 (SP – 4)
(0xFFE0)	0xFFB4 (SP + 0)
(0xFFEC)	0xFFB8 (SP + 4)
i (0)	0xFFBC (SP + 8)
j0 (2)	0xFFC0 (SP + 12)
j1 (5)	0xFFC4 (SP + 16)
(ret_addr)	0xFFC8 (SP + 20)
a (0xFFD8)	0xFFCC (SP + 24)
n (10)	0xFFD0 (SP + 28)
rand_seed (3)	0xFFD4 (SP + 32)
a[0] (0)	0xFFD8
a[1] (1)	0xFFDC
a[2] (2)	0xFFE0
a[3] (3)	0xFFE4
a[4] (4)	0xFFE8
a[5] (5)	0xFFEC
a[6] (6)	0xFFF0
a[7] (7)	0xFFF4
a[8] (8)	0xFFF8
a[9] (9)	0xFFFC
	0x10000

Stack Frame “swap”

```
void swap(int * a0, int * a1)
{
        int t = *a0;
        *a0 = *a1;
        *a1 = t;
}
```



SP = 0xFFAC

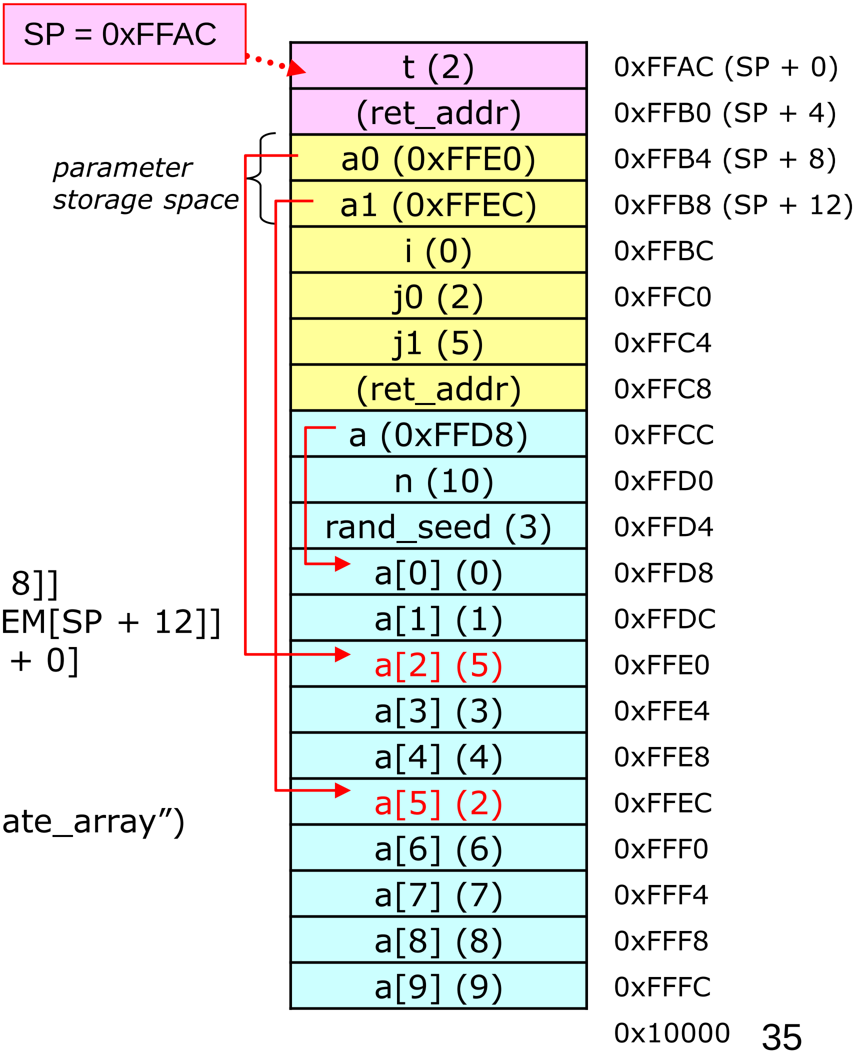
parameter storage space

t	0xFFAC (SP + 0)
(ret_addr)	0xFFB0 (SP + 4)
a0 (0xFFE0)	0xFFB4 (SP + 8)
a1 (0xFFEC)	0xFFB8 (SP + 12)
i (0)	0xFFBC
j0 (2)	0xFFC0
j1 (5)	0xFFC4
(ret_addr)	0xFFC8
a (0xFFD8)	0xFFCC
n (10)	0xFFD0
rand_seed (3)	0xFFD4
a[0] (0)	0xFFD8
a[1] (1)	0xFFDC
a[2] (2)	0xFFE0
a[3] (3)	0xFFE4
a[4] (4)	0xFFE8
a[5] (5)	0xFFEC
a[6] (6)	0xFFF0
a[7] (7)	0xFFF4
a[8] (8)	0xFFF8
a[9] (9)	0xFFFC
	0x10000

- Enter “swap”:
  - SP = SP – 8 = 0xFFAC
  - 4 bytes for storing “int t” : &(t) = SP + 0
  - 4 bytes for storing ret_addr to the caller
- Parameter address
  - SP + 8 : int * a0 (MEM[SP + 8] = 0xFFE0)
  - SP + 12 : int * a1 (MEM[SP + 12] = 0xFFEC)

Stack Frame “swap2”

```
void swap(int * a0, int * a1)
{
        int t = *a0;
        *a0 = *a1;
        *a1 = t;
}
```



SP = 0xFFAC

parameter storage space

t (2)	0xFFAC (SP + 0)
(ret_addr)	0xFFB0 (SP + 4)
a0 (0xFFE0)	0xFFB4 (SP + 8)
a1 (0xFFEC)	0xFFB8 (SP + 12)
i (0)	0xFFBC
j0 (2)	0xFFC0
j1 (5)	0xFFC4
(ret_addr)	0xFFC8
a (0xFFD8)	0xFFCC
n (10)	0xFFD0
rand_seed (3)	0xFFD4
a[0] (0)	0xFFD8
a[1] (1)	0xFFDC
a[2] (5)	0xFFE0
a[3] (3)	0xFFE4
a[4] (4)	0xFFE8
a[5] (2)	0xFFEC
a[6] (6)	0xFFF0
a[7] (7)	0xFFF4
a[8] (8)	0xFFF8
a[9] (9)	0xFFFC
	0x10000

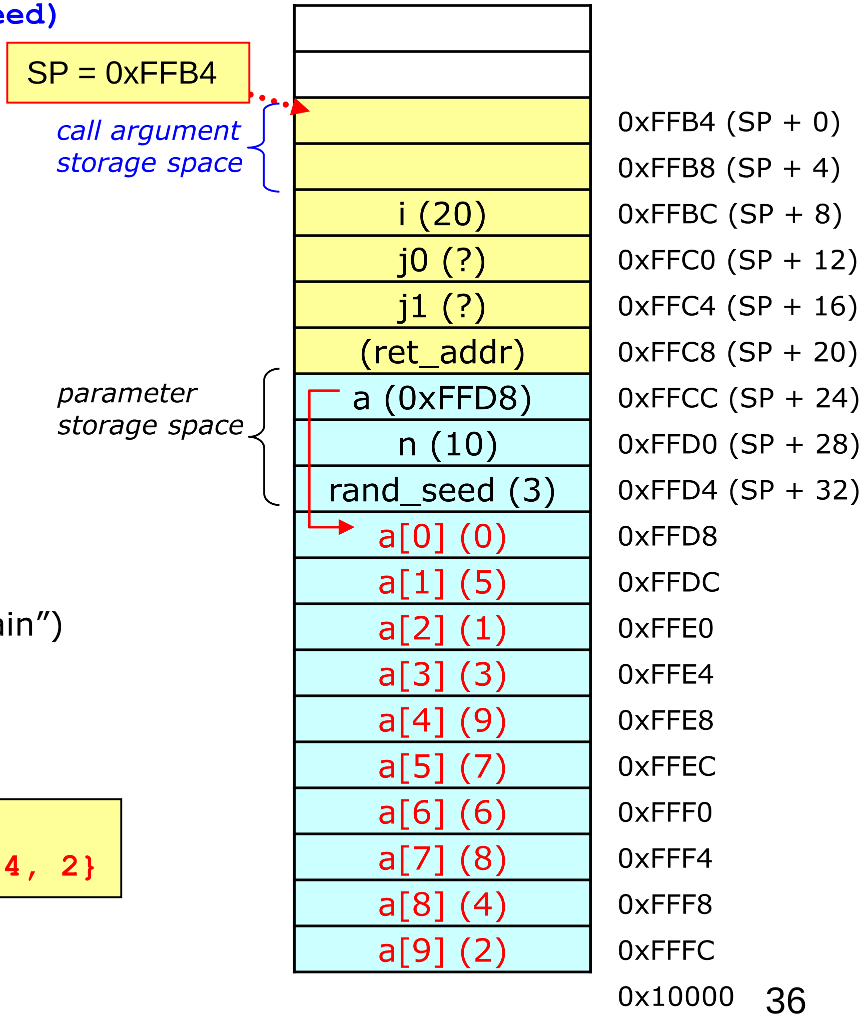
- Operations
  - `t = *a0` : MEM[SP + 0] ← MEM[MEM[SP + 8]]
  - `*a0 = *a1` : MEM[MEM[SP + 8]] ← MEM[MEM[SP + 12]]
  - `*a1 = t` : MEM[MEM[SP + 12]] ← MEM[SP + 0]
- Exit “swap2”:
  - SP = SP + 8 = 0xFFB4
  - ret_addr = MEM[SP – 4] = MEM[0xFFB0]
  - next_instr = I_MEM[ret_addr] (back to “create_array”)

Stack Frame “create_array”

```
void create_array (int a[], int n, int rand_seed)
{
        int i;
        for(i = 0; i < n; i ++)
            a[i] = i;
        srand(rand_seed);
        for(i = 0; i < 2 * n; i ++){
            int j0 = rand() % n;
            int j1 = rand() % n;
            swap(&a[j0], &a[j1]);
        }
}
```

- Exit “create_array”:
    - SP = SP + 24 = 0xFFCC
    - ret_addr = MEM[SP – 4] = MEM[0xFFC8]
    - next_instr = I_MEM[ret_addr] (back to “main”)

```
Initial array:
a[0:9] = {0, 5, 1, 3, 9, 7, 6, 8, 4, 2}
```



SP = 0xFFB4

call argument storage space

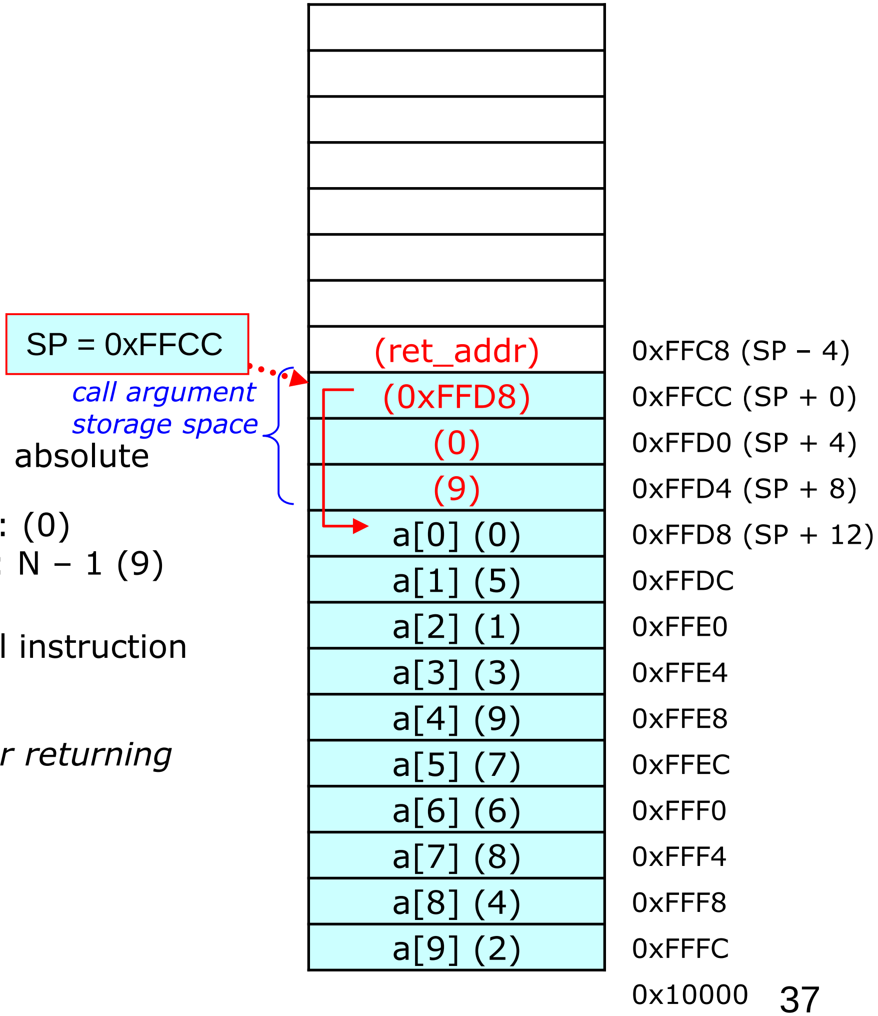
parameter storage space

	0xFFB4 (SP + 0)
	0xFFB8 (SP + 4)
i (20)	0xFFBC (SP + 8)
j0 (?)	0xFFC0 (SP + 12)
j1 (?)	0xFFC4 (SP + 16)
(ret_addr)	0xFFC8 (SP + 20)
a (0xFFD8)	0xFFCC (SP + 24)
n (10)	0xFFD0 (SP + 28)
rand_seed (3)	0xFFD4 (SP + 32)
a[0] (0)	0xFFD8
a[1] (5)	0xFFDC
a[2] (1)	0xFFE0
a[3] (3)	0xFFE4
a[4] (9)	0xFFE8
a[5] (7)	0xFFEC
a[6] (6)	0xFFF0
a[7] (8)	0xFFF4
a[8] (4)	0xFFF8
a[9] (2)	0xFFFC
	0x10000

Stack Frame “main”

```
#define N 10
#define RSEED 3
void main()
{
      int a[N]; /* size = 4N = 40 bytes */
      create_array(a, N, RSEED);
      print_array2(a, 0, N - 1);
      quick_sort(a, 0, N - 1);
      print_array2(a, 0, N - 1);
}
```

- Before calling “print_array2”:
  - Copy 1st argument value to MEM[SP + 0] : absolute address of “a” (SP + 12 = 0xFFD8)
  - Copy 2nd argument value to MEM[SP + 4] : (0)
  - Copy 3rd argument value to MEM[SP + 8] : N – 1 (9)
- Call “print_array2”:
  - Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - Jump to “print_array2”
  - ← *Program resumes at I_MEM[ret_addr] after returning from “print_array2”*



SP = 0xFFCC

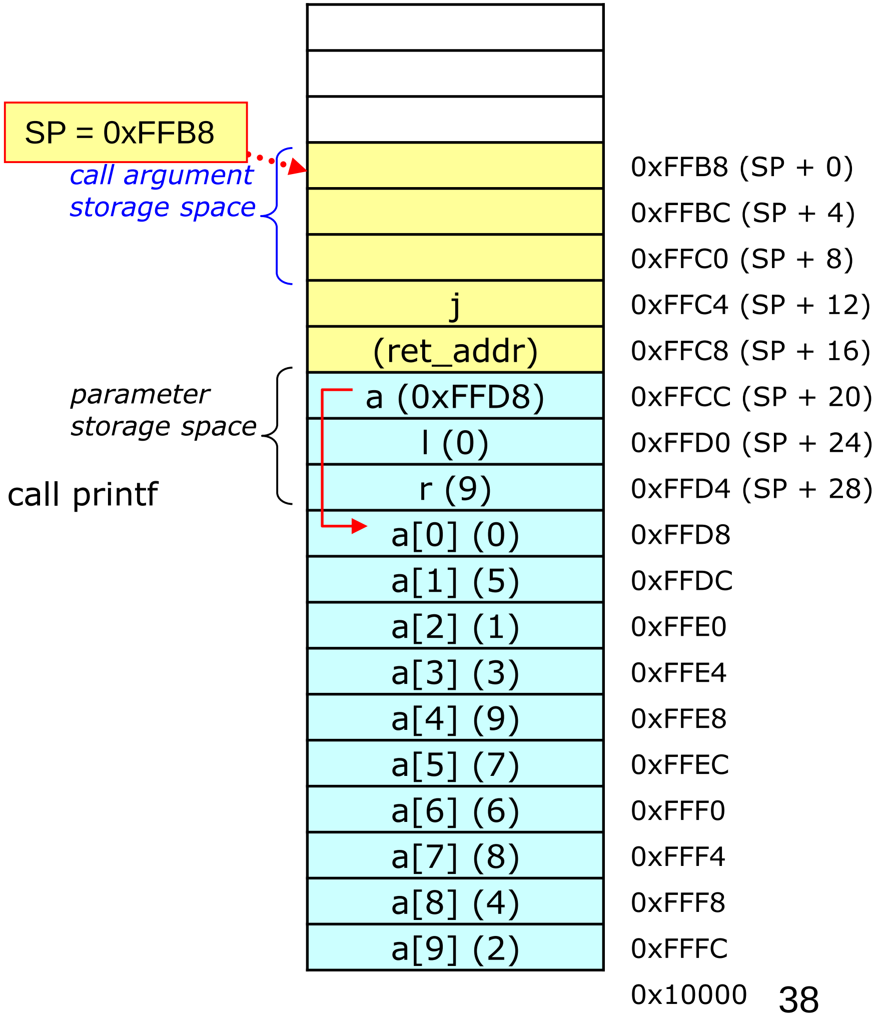
call argument storage space

(ret_addr)	0xFFC8 (SP – 4)
(0xFFD8)	0xFFCC (SP + 0)
(0)	0xFFD0 (SP + 4)
(9)	0xFFD4 (SP + 8)
a[0] (0)	0xFFD8 (SP + 12)
a[1] (5)	0xFFDC
a[2] (1)	0xFFE0
a[3] (3)	0xFFE4
a[4] (9)	0xFFE8
a[5] (7)	0xFFEC
a[6] (6)	0xFFF0
a[7] (8)	0xFFF4
a[8] (4)	0xFFF8
a[9] (2)	0xFFFC
	0x10000

Stack Frame “print_array2”

```
void print_array2(int a[], int l, int r)
{
        int j;
        printf("a[%d:%d] = {", l, r);
        for(j = l; j <= r; j ++){
            if(j > l) printf(", ");
            printf("%d", a[j]);
        }
        printf("}\n");
}
```

- Enter “print_array2”:
  - SP = SP – 20 = 0xFFB8
  - 12 bytes for storing 3 argument values for call printf
  - 4 bytes for storing “int j” : &(j) = SP + 12
  - 4 bytes for storing ret_addr to the caller
- Parameter address
  - SP + 20 : int a[]
  - SP + 24 : int l
  - SP + 28 : int r

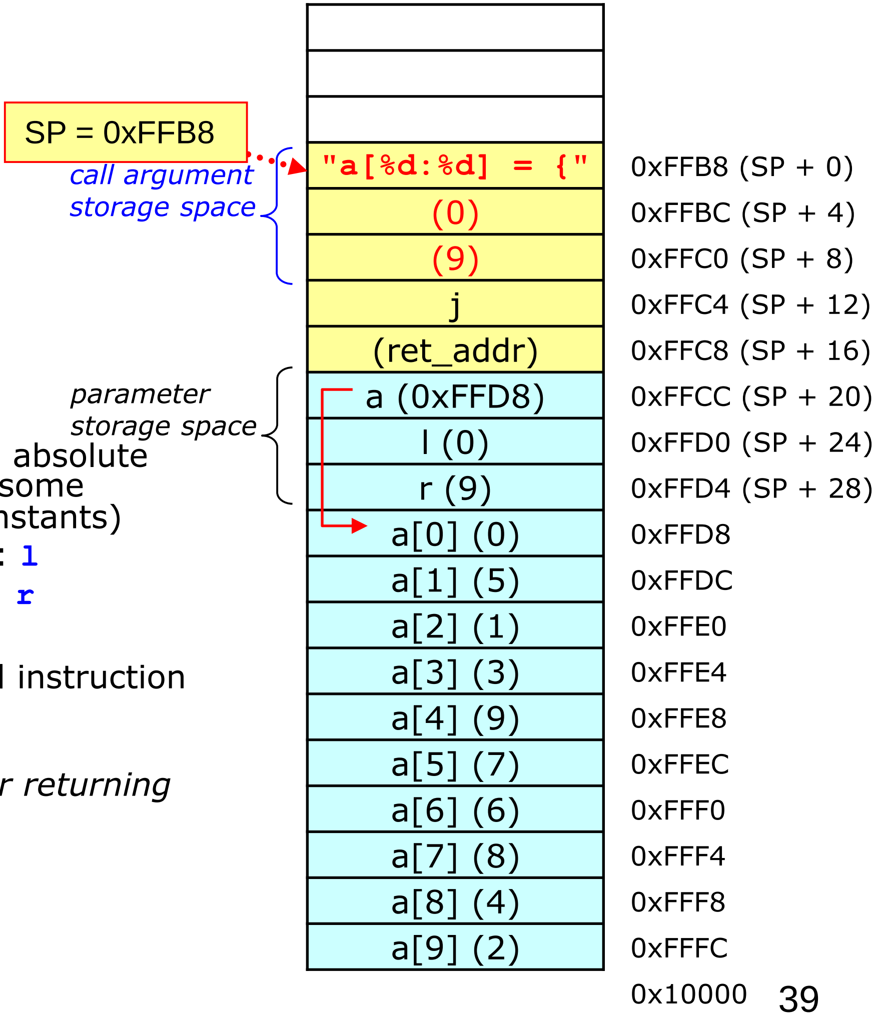


Stack Frame “print_array2”

```
void print_array2(int a[], int l, int r)
{
        int j;
        printf("a[%d:%d] = {", l, r);
        for(j = l; j <= r; j ++){
            if(j > l) printf(", ");
            printf("%d", a[j]);
        }
        printf("}\n");
}
```

- Before calling 1st “printf”:
  - Copy 1st argument value to MEM[SP + 0] : absolute address of string literal `"a[%d:%d] = {"` (some address in memory that stores array of constants)
  - Copy 2nd argument value to MEM[SP + 4] : `l`
  - Copy 3rd argument value to MEM[SP + 8] : `r`
- Call “printf”:
  - Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - Jump to “printf”
  - ← *Program resumes at I_MEM[ret_addr] after returning from “printf”*

```
Output:
a[0:9] = {
```



SP = 0xFFB8

call argument storage space

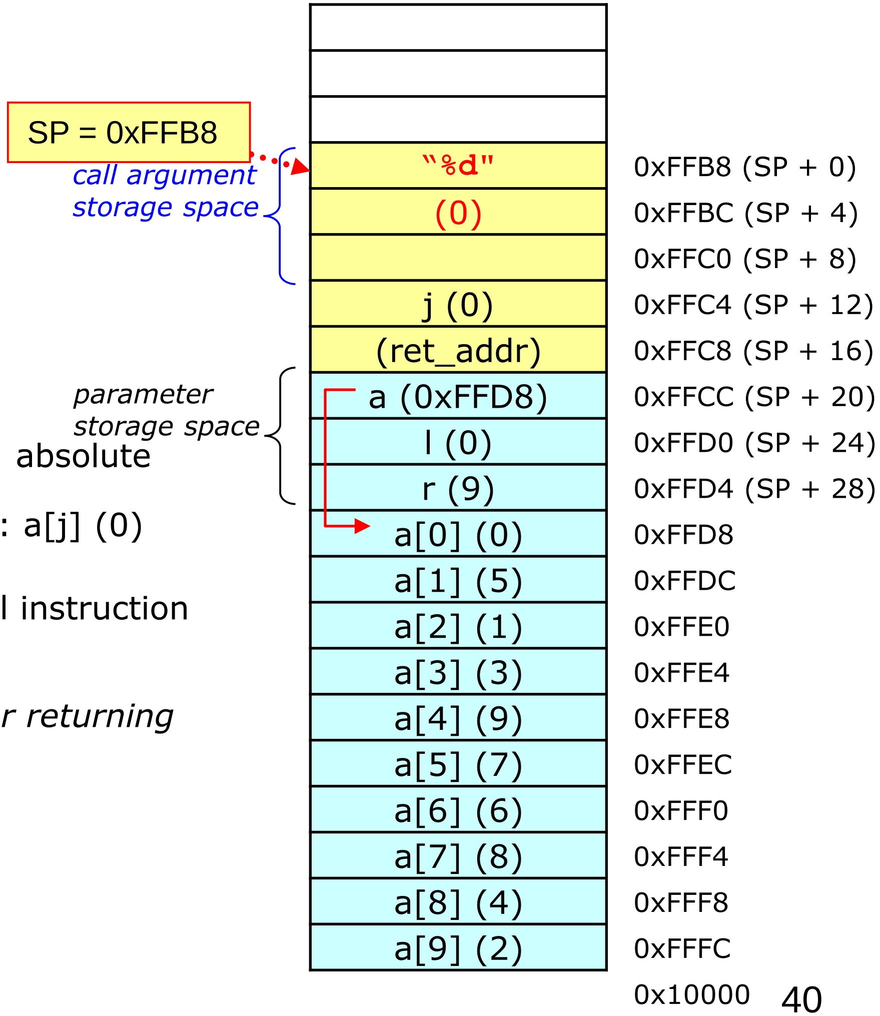
parameter storage space

`"a[%d:%d] = {"`	0xFFB8 (SP + 0)
(0)	0xFFBC (SP + 4)
(9)	0xFFC0 (SP + 8)
j	0xFFC4 (SP + 12)
(ret_addr)	0xFFC8 (SP + 16)
a (0xFFD8)	0xFFCC (SP + 20)
l (0)	0xFFD0 (SP + 24)
r (9)	0xFFD4 (SP + 28)
a[0] (0)	0xFFD8
a[1] (5)	0xFFDC
a[2] (1)	0xFFE0
a[3] (3)	0xFFE4
a[4] (9)	0xFFE8
a[5] (7)	0xFFEC
a[6] (6)	0xFFF0
a[7] (8)	0xFFF4
a[8] (4)	0xFFF8
a[9] (2)	0xFFFC
	0x10000

Stack Frame “print_array2”

```
void print_array2(int a[], int l, int r)
{
        int j;
        printf("a[%d:%d] = {", l, r);
        for(j = l; j <= r; j ++){
            if(j > l) printf(", ");
            printf("%d", a[j]);
        }
        printf("}\n");
}
```

- Before calling 3rd “printf”:
  - Copy 1st argument value to MEM[SP + 0] : absolute address of string literal “%d"
  - Copy 2nd argument value to MEM[SP + 4] : a[j] (0)
- Call “printf”:
  - Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - Jump to “printf”
  - ← *Program resumes at I_MEM[ret_addr] after returning from “printf”*



SP = 0xFFB8

call argument storage space

parameter storage space

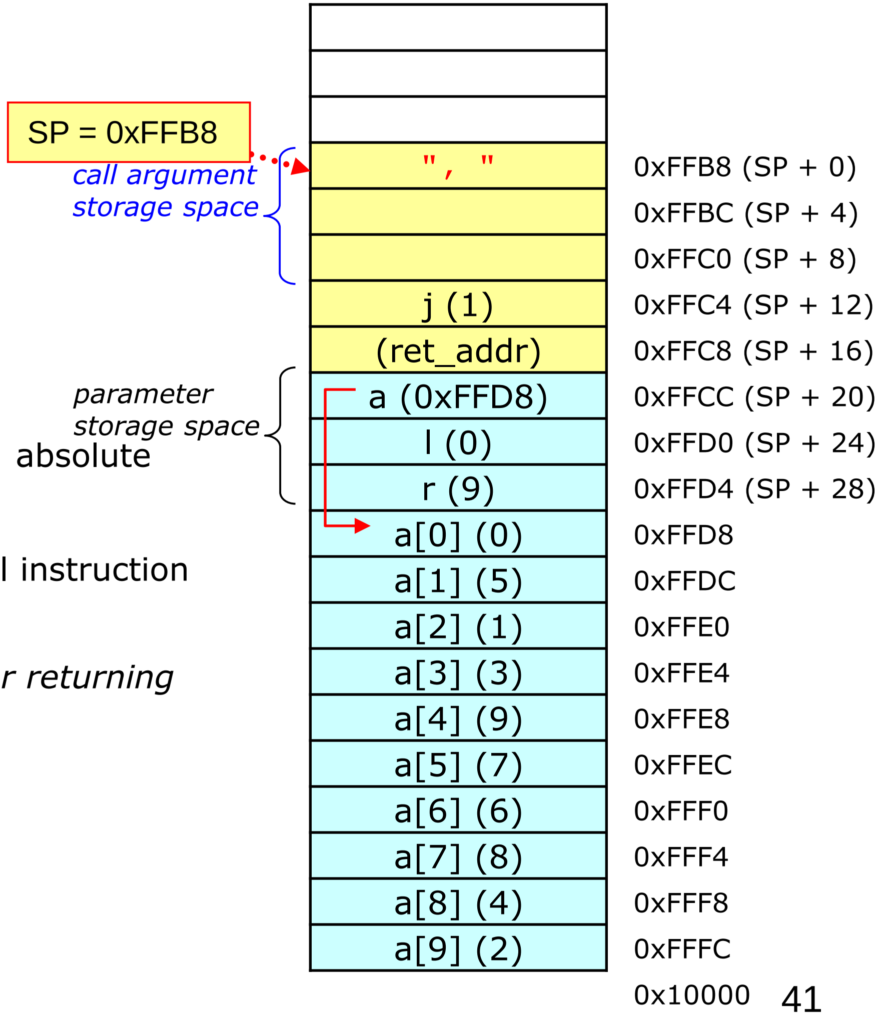
“%d"	0xFFB8 (SP + 0)
(0)	0xFFBC (SP + 4)
	0xFFC0 (SP + 8)
j (0)	0xFFC4 (SP + 12)
(ret_addr)	0xFFC8 (SP + 16)
a (0xFFD8)	0xFFCC (SP + 20)
l (0)	0xFFD0 (SP + 24)
r (9)	0xFFD4 (SP + 28)
a[0] (0)	0xFFD8
a[1] (5)	0xFFDC
a[2] (1)	0xFFE0
a[3] (3)	0xFFE4
a[4] (9)	0xFFE8
a[5] (7)	0xFFEC
a[6] (6)	0xFFF0
a[7] (8)	0xFFF4
a[8] (4)	0xFFF8
a[9] (2)	0xFFFC
	0x10000

```
Output:
a[0:9] = {0
```

Stack Frame “print_array2”

```
void print_array2(int a[], int l, int r)
{
        int j;
        printf("a[%d:%d] = {", l, r);
        for(j = l; j <= r; j ++){
            if(j > l) printf(", ");
            printf("%d", a[j]);
        }
        printf("}\n");
}
```

- Before calling 2nd “printf”:
  - Copy 1st argument value to MEM[SP + 0] : absolute address of string literal `", "`
- Call “printf”:
  - Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - Jump to “printf”
  - ← *Program resumes at I_MEM[ret_addr] after returning from “printf”*



SP = 0xFFB8

call argument storage space

parameter storage space

", " 0xFFB8 (SP + 0)

0xFFBC (SP + 4)

0xFFC0 (SP + 8)

j (1) 0xFFC4 (SP + 12)

(ret_addr) 0xFFC8 (SP + 16)

a (0xFFD8) 0xFFCC (SP + 20)

l (0) 0xFFD0 (SP + 24)

r (9) 0xFFD4 (SP + 28)

a[0] (0) 0xFFD8

a[1] (5) 0xFFDC

a[2] (1) 0xFFE0

a[3] (3) 0xFFE4

a[4] (9) 0xFFE8

a[5] (7) 0xFFEC

a[6] (6) 0xFFF0

a[7] (8) 0xFFF4

a[8] (4) 0xFFF8

a[9] (2) 0xFFFC

0x10000

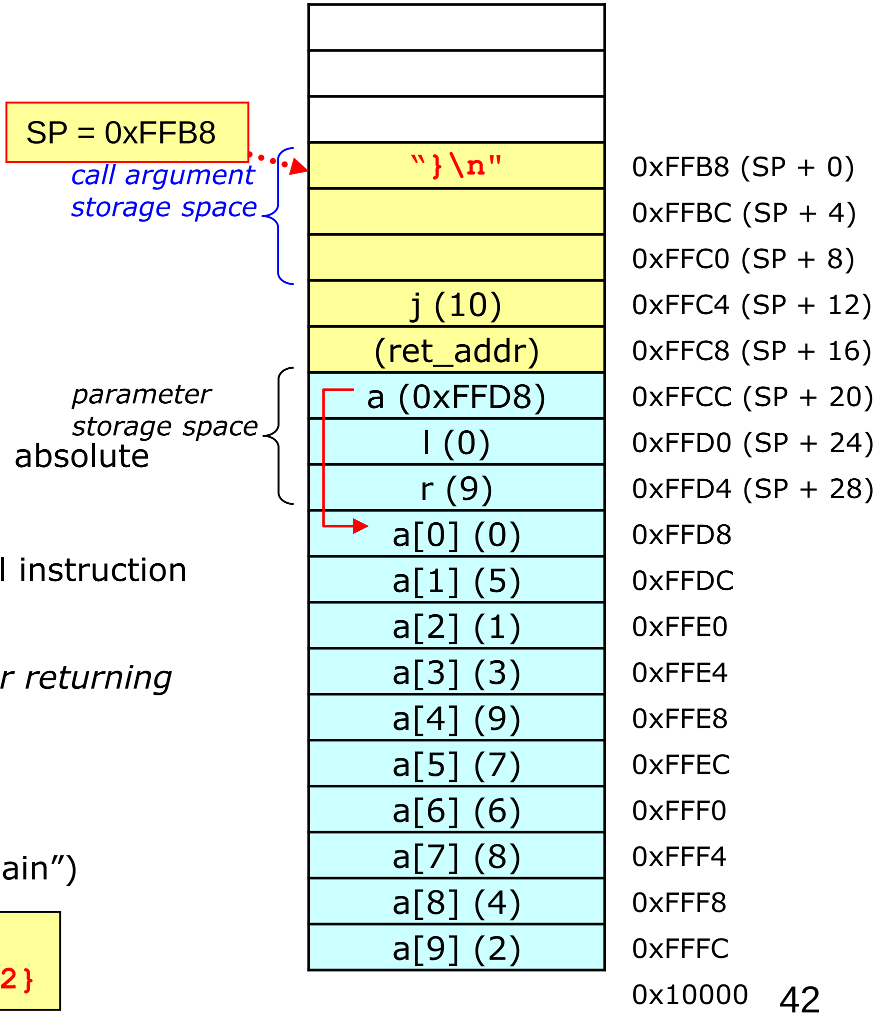
```
Output:
a[0:9] = {0,
```

Stack Frame “print_array2”

```
void print_array2(int a[], int l, int r)
{
        int j;
        printf("a[%d:%d] = {", l, r);
        for(j = l; j <= r; j ++){
            if(j > l) printf(", ");
            printf("%d", a[j]);
        }
        printf("}\n");
}
```

- Before calling 4th “printf”:
  - Copy 1st argument value to MEM[SP + 0] : absolute address of string literal “}\n"
- Call “printf”:
  - Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - Jump to “printf”
  - ← *Program resumes at I_MEM[ret_addr] after returning from “printf”*
- Exit “print_array2”:
  - SP = SP + 20 = 0xFFCC
  - ret_addr = MEM[SP – 4] = MEM[0xFFC8]
  - next_instr = I_MEM[ret_addr] (back to “main”)

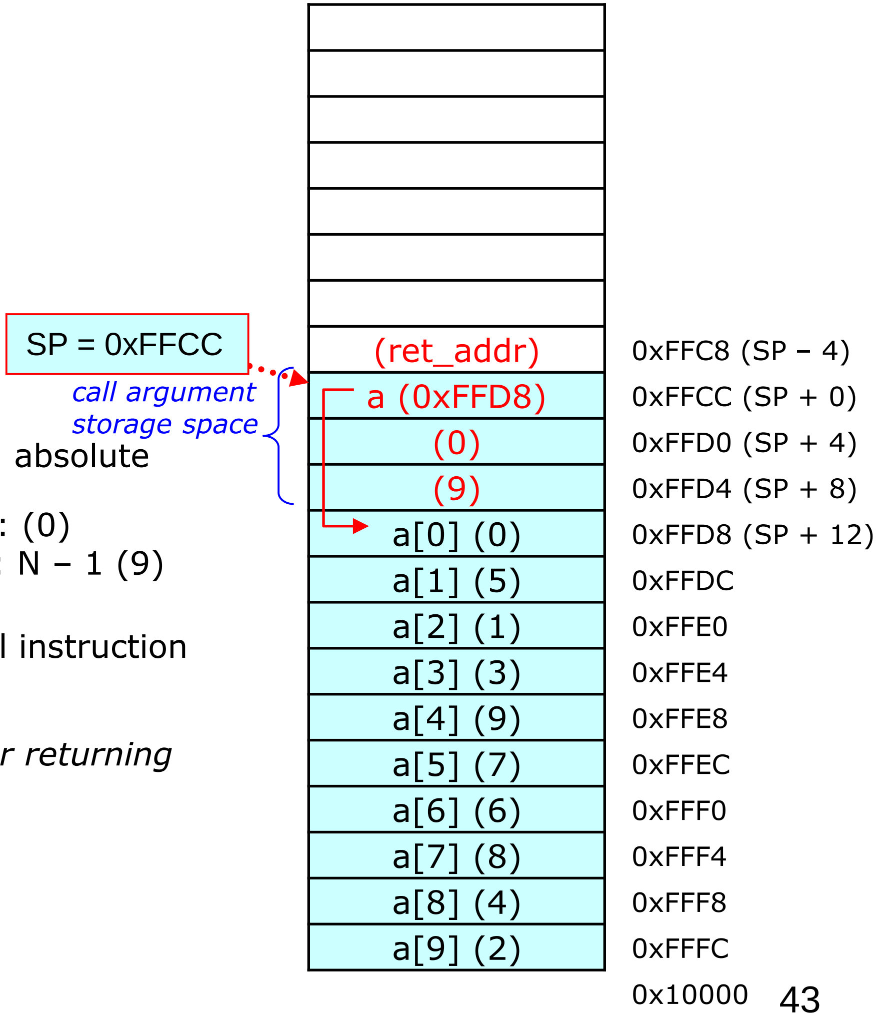
```
Output:
a[0:9] = {0, 5, 1, 3, 9, 7, 6, 8, 4, 2}
```



Stack Frame “main”

```
#define N 10
#define RSEED 3
void main()
{
        int a[N]; /* size = 4N = 40 bytes */
        create_array(a, N, RSEED);
        print_array2(a, 0, N - 1);
        quick_sort(a, 0, N – 1);
        print_array2(a, 0, N - 1);
}
```

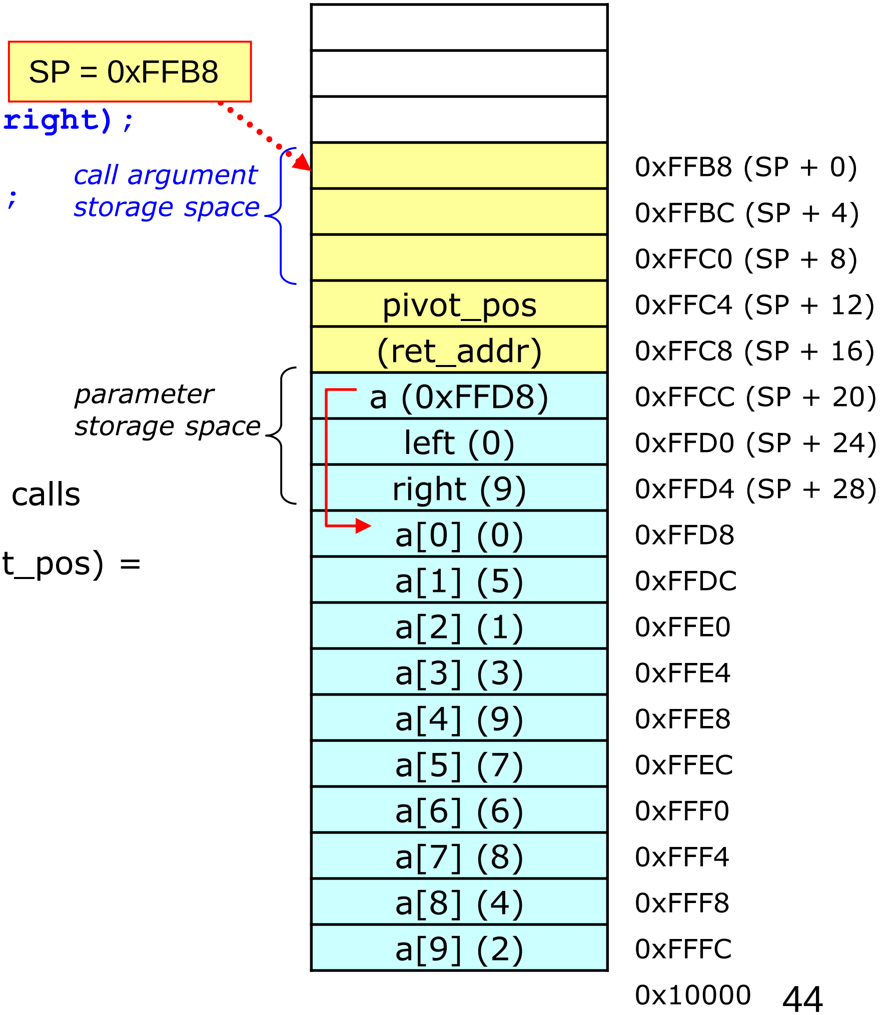
- Before calling “quick_sort”:
  - Copy 1st argument value to MEM[SP + 0] : absolute address of “a” (SP + 12 = 0xFFD8)
  - Copy 2nd argument value to MEM[SP + 4] : (0)
  - Copy 3rd argument value to MEM[SP + 8] : N – 1 (9)
- Call “quick_sort”:
  - Copy next instruction address after the call instruction (ret_addr) to MEM[SP – 4]
  - Jump to “quick_sort”
  - ← *Program resumes at I_MEM[ret_addr] after returning from “quick_sort”*



Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```

- Enter “quick_sort”:
  - SP = SP – 20 = 0xFFB8
  - 12 bytes for storing 3 argument values for calls **partition** and **quick_sort**
  - 4 bytes for storing “int pivot_pos” : &(pivot_pos) = SP + 12
  - 4 bytes for storing ret_addr to the caller
- Parameter address
  - SP + 20 : int a[] (0xFFD8)
  - SP + 24 : int left (0)
  - SP + 28 : int right (9)



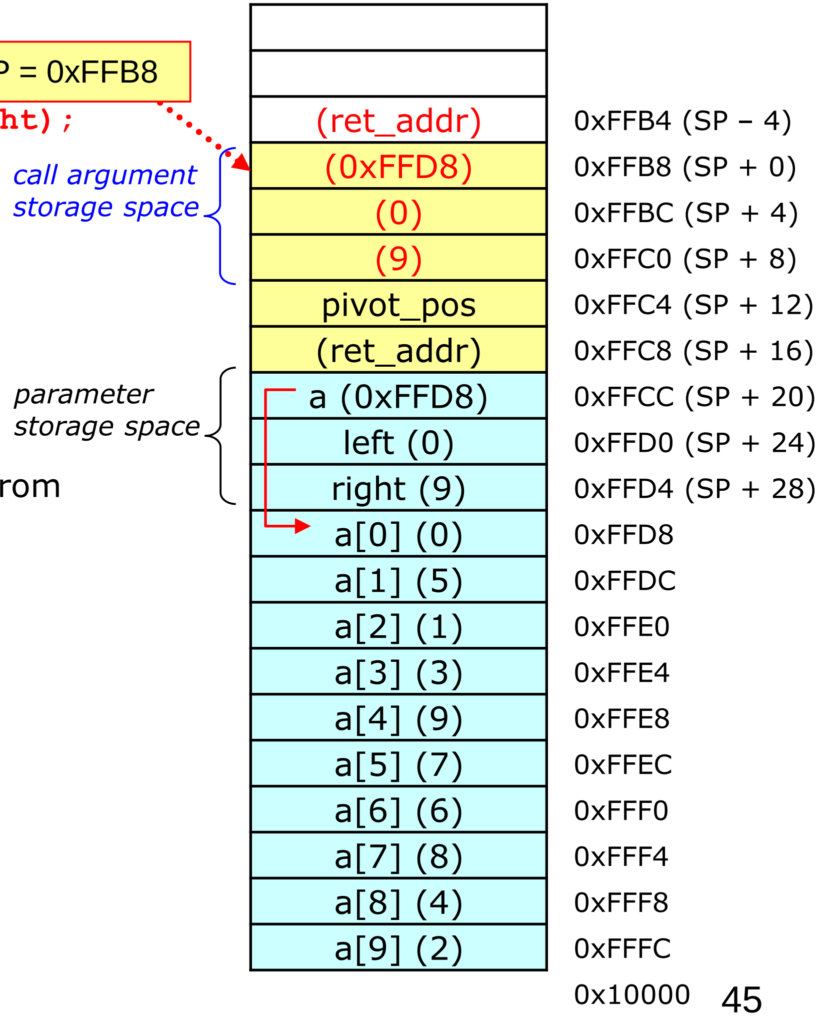
SP = 0xFFB8

call argument storage space		0xFFB8 (SP + 0)
		0xFFBC (SP + 4)
		0xFFC0 (SP + 8)
	pivot_pos	0xFFC4 (SP + 12)
	(ret_addr)	0xFFC8 (SP + 16)
parameter storage space	a (0xFFD8)	0xFFCC (SP + 20)
	left (0)	0xFFD0 (SP + 24)
	right (9)	0xFFD4 (SP + 28)
	a[0] (0)	0xFFD8
	a[1] (5)	0xFFDC
	a[2] (1)	0xFFE0
	a[3] (3)	0xFFE4
	a[4] (9)	0xFFE8
	a[5] (7)	0xFFEC
	a[6] (6)	0xFFF0
	a[7] (8)	0xFFF4
	a[8] (4)	0xFFF8
	a[9] (2)	0xFFFC
		0x10000

Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```

SP = 0xFFB8



(ret_addr)	0xFFB4 (SP – 4)
(0xFFD8)	0xFFB8 (SP + 0)
(0)	0xFFBC (SP + 4)
(9)	0xFFC0 (SP + 8)
pivot_pos	0xFFC4 (SP + 12)
(ret_addr)	0xFFC8 (SP + 16)
a (0xFFD8)	0xFFCC (SP + 20)
left (0)	0xFFD0 (SP + 24)
right (9)	0xFFD4 (SP + 28)
a[0] (0)	0xFFD8
a[1] (5)	0xFFDC
a[2] (1)	0xFFE0
a[3] (3)	0xFFE4
a[4] (9)	0xFFE8
a[5] (7)	0xFFEC
a[6] (6)	0xFFF0
a[7] (8)	0xFFF4
a[8] (4)	0xFFF8
a[9] (2)	0xFFFC
	0x10000

call argument storage space

parameter storage space

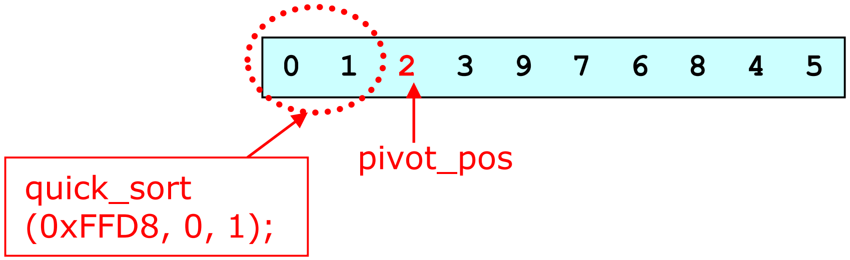
- CALL partition(0xFFD8, 0, 9):
  - (ret_val) stored at MEM[SP + 0] after return from “partition”

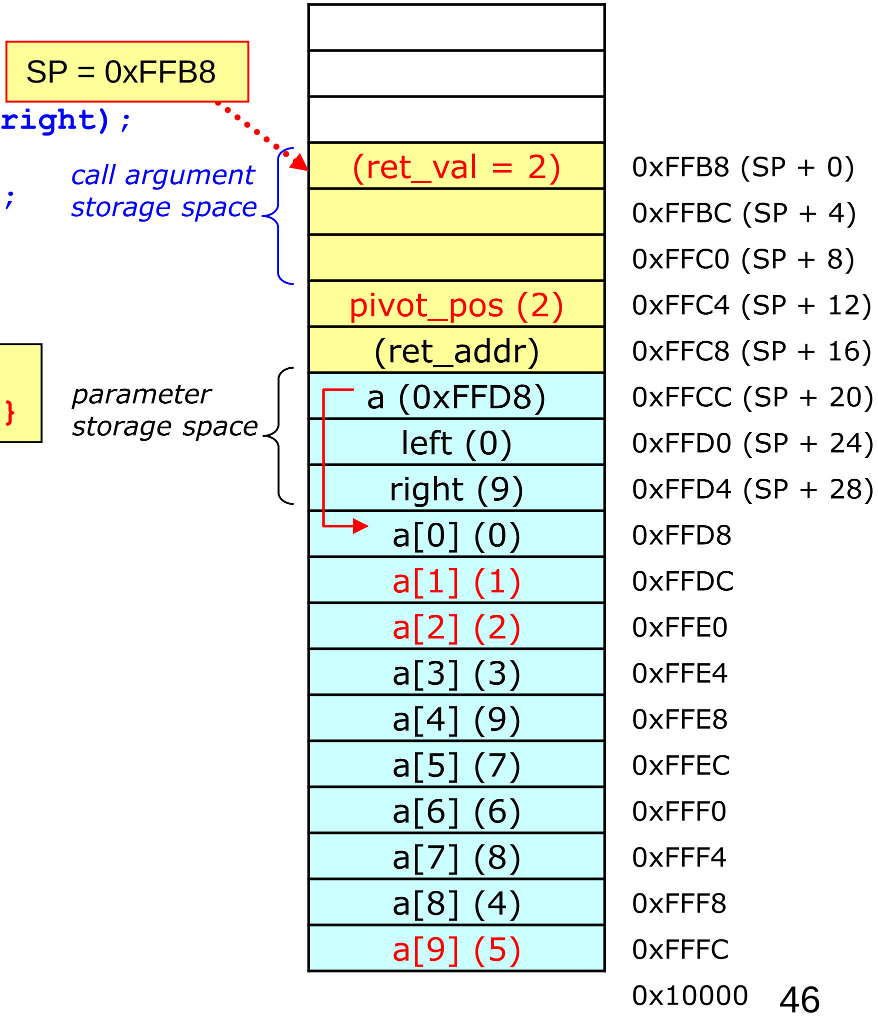
Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```

```
Output:
pivot(2): a[0:9] = {0, 1, 2, 3, 9, 7, 6, 8, 4, 5}
```

- **pivot_pos = ret_val = 2**
  MEM[SP + 12] ← MEM[SP + 0]
- CALL quick_sort(0xFFD8, 0, 1):





SP = 0xFFB8

call argument storage space

parameter storage space

(ret_val = 2)	0xFFB8 (SP + 0)
	0xFFBC (SP + 4)
	0xFFC0 (SP + 8)
pivot_pos (2)	0xFFC4 (SP + 12)
(ret_addr)	0xFFC8 (SP + 16)
a (0xFFD8)	0xFFCC (SP + 20)
left (0)	0xFFD0 (SP + 24)
right (9)	0xFFD4 (SP + 28)
a[0] (0)	0xFFD8
a[1] (1)	0xFFDC
a[2] (2)	0xFFE0
a[3] (3)	0xFFE4
a[4] (9)	0xFFE8
a[5] (7)	0xFFEC
a[6] (6)	0xFFF0
a[7] (8)	0xFFF4
a[8] (4)	0xFFF8
a[9] (5)	0xFFFC
	0x10000

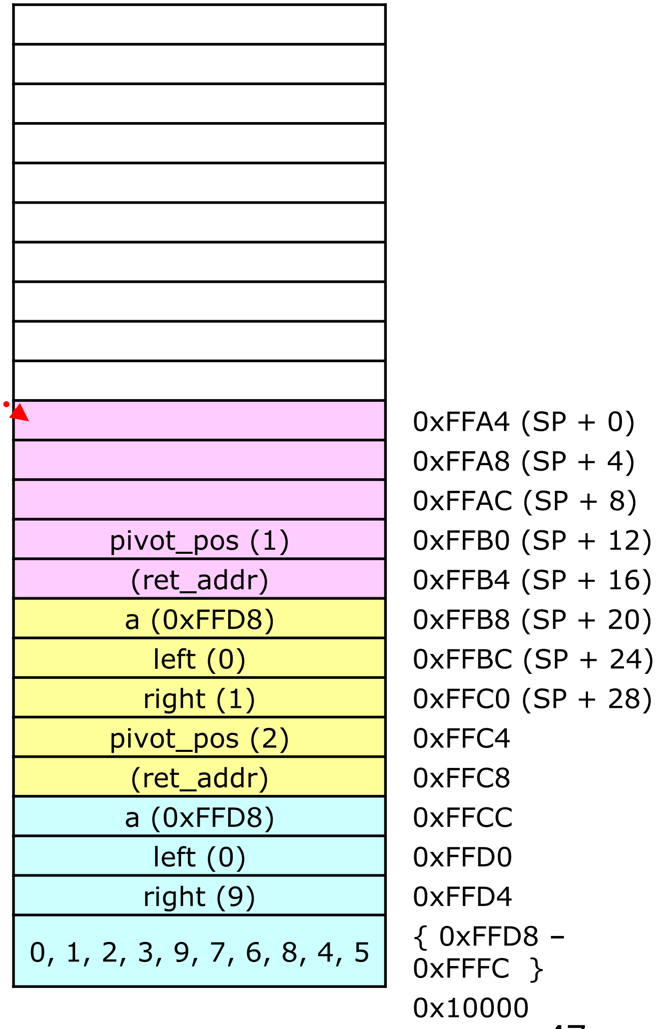
Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```

SP = 0xFFA4

call argument storage space

parameter storage space



	0xFFA4 (SP + 0)
	0xFFA8 (SP + 4)
	0xFFAC (SP + 8)
pivot_pos (1)	0xFFB0 (SP + 12)
(ret_addr)	0xFFB4 (SP + 16)
a (0xFFD8)	0xFFB8 (SP + 20)
left (0)	0xFFBC (SP + 24)
right (1)	0xFFC0 (SP + 28)
pivot_pos (2)	0xFFC4
(ret_addr)	0xFFC8
a (0xFFD8)	0xFFCC
left (0)	0xFFD0
right (9)	0xFFD4
0, 1, 2, 3, 9, 7, 6, 8, 4, 5	{ 0xFFD8 – 0xFFFC }
	0x10000

- Enter quick_sort(a = 0xFFD8, left = 0, right = 1):
  - SP = SP – 20 = 0xFFA4
  - a[0:1] = {0, 1}
- CALL partition(0xFFD8, 0, 1):

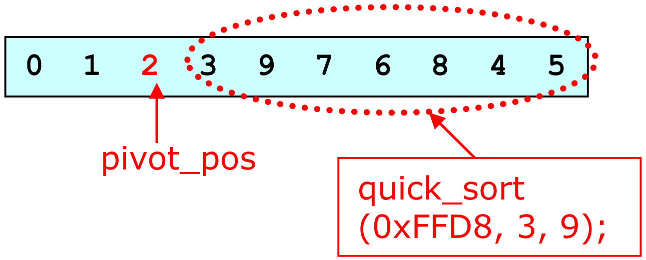
```
Output:
pivot(1): a[0:1] = {0, 1}
```

- CALL quick_sort(0xFFD8, 0, 0):
  - left = right = 0 : so recursion is skipped
- CALL quick_sort(0xFFD8, 2, 1):
  - left > right : so recursion is skipped
- Exit quick_sort:
  - SP = SP + 20 = 0xFFB8

Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```

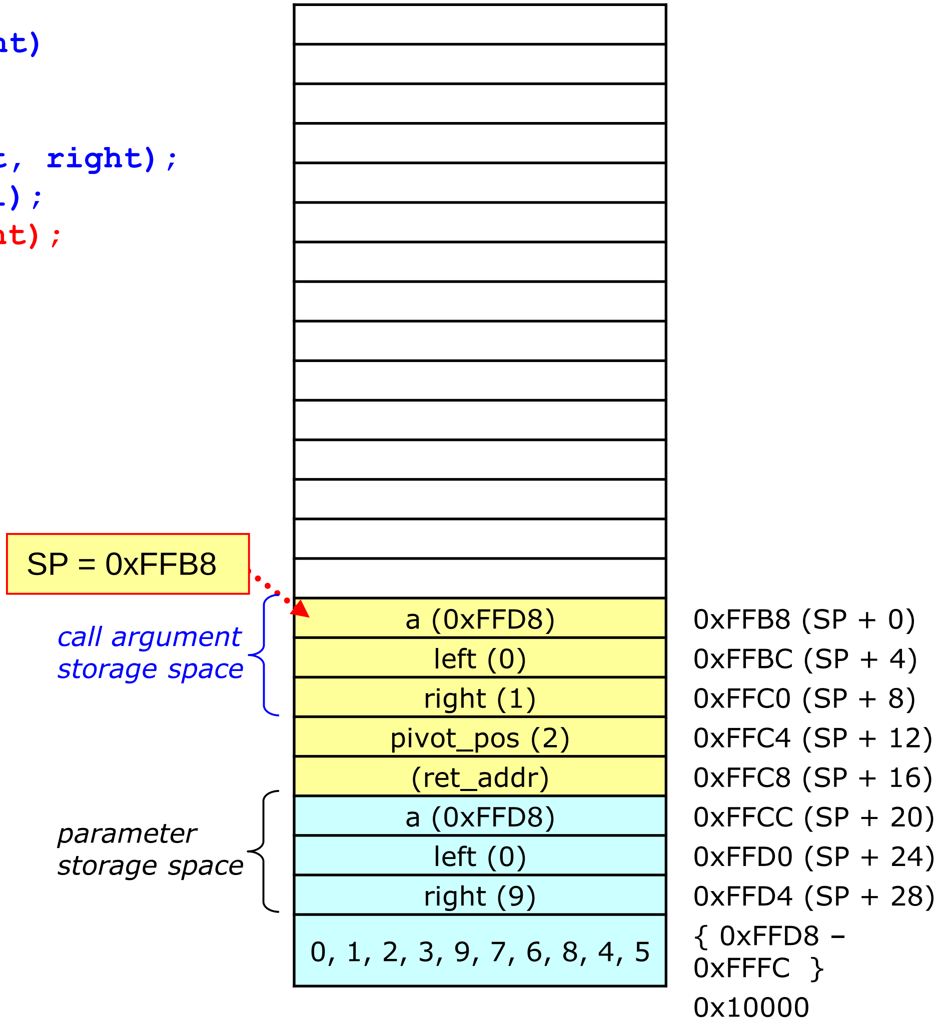
- CALL quick_sort(0xFFD8, 3, 9):



0 1 2 3 9 7 6 8 4 5

pivot_pos

quick_sort (0xFFD8, 3, 9);



SP = 0xFFB8

call argument storage space

parameter storage space

a (0xFFD8)	0xFFB8 (SP + 0)
left (0)	0xFFBC (SP + 4)
right (1)	0xFFC0 (SP + 8)
pivot_pos (2)	0xFFC4 (SP + 12)
(ret_addr)	0xFFC8 (SP + 16)
a (0xFFD8)	0xFFCC (SP + 20)
left (0)	0xFFD0 (SP + 24)
right (9)	0xFFD4 (SP + 28)
0, 1, 2, 3, 9, 7, 6, 8, 4, 5	{ 0xFFD8 – 0xFFFC }
	0x10000

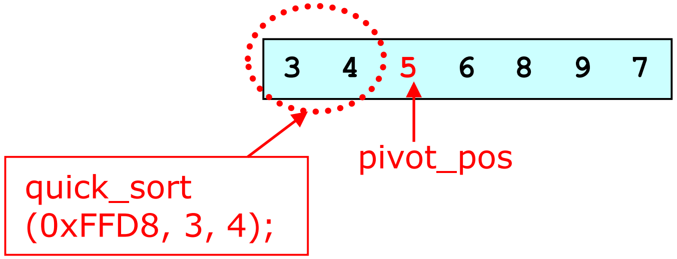
Stack Frame “quick_sort”

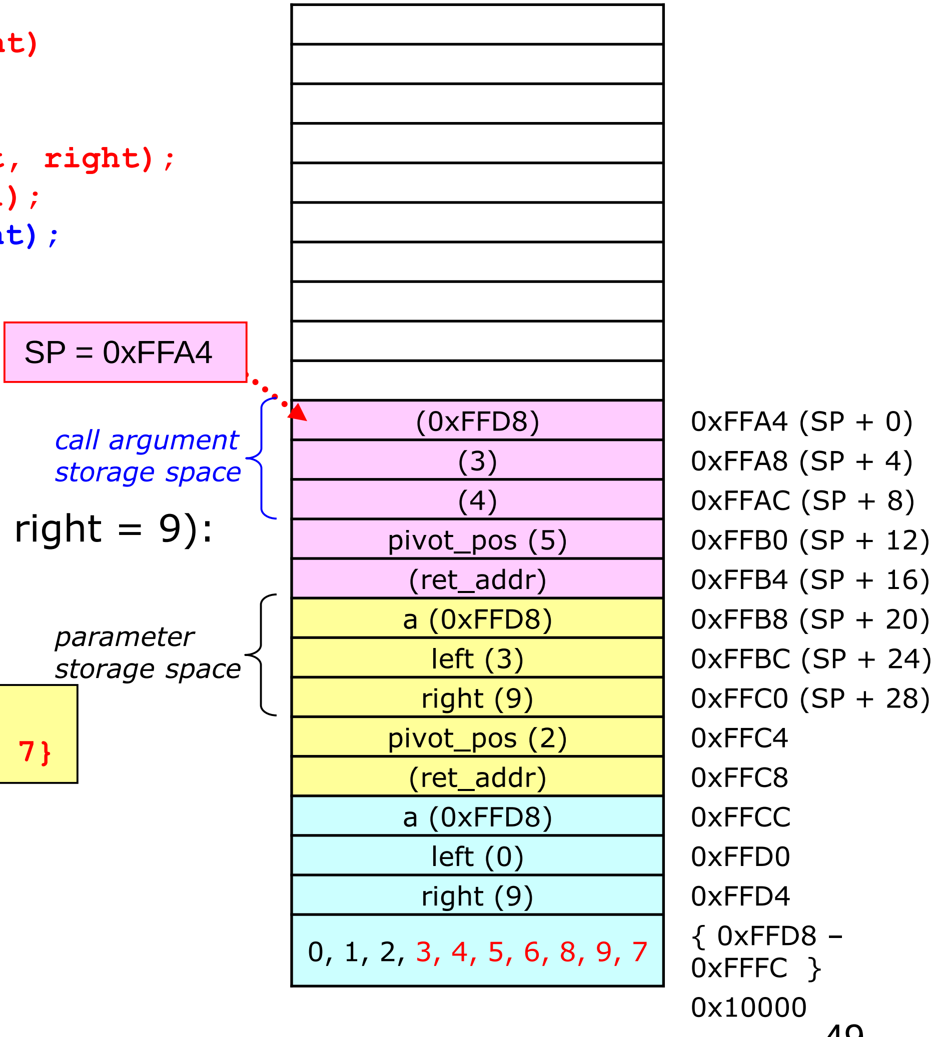
```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```

- Enter quick_sort(a = 0xFFD8, left = 3, right = 9):
  - SP = SP – 20 = 0xFFA4
  - a[3:9] = {3, 9, 7, 6, 8, 4, 5}
- CALL partition(0xFFD8, 3, 9):

```
Output:
pivot(5): a[3:9] = {3, 4, 5, 6, 8, 9, 7}
```

- CALL quick_sort(0xFFD8, 3, 4):

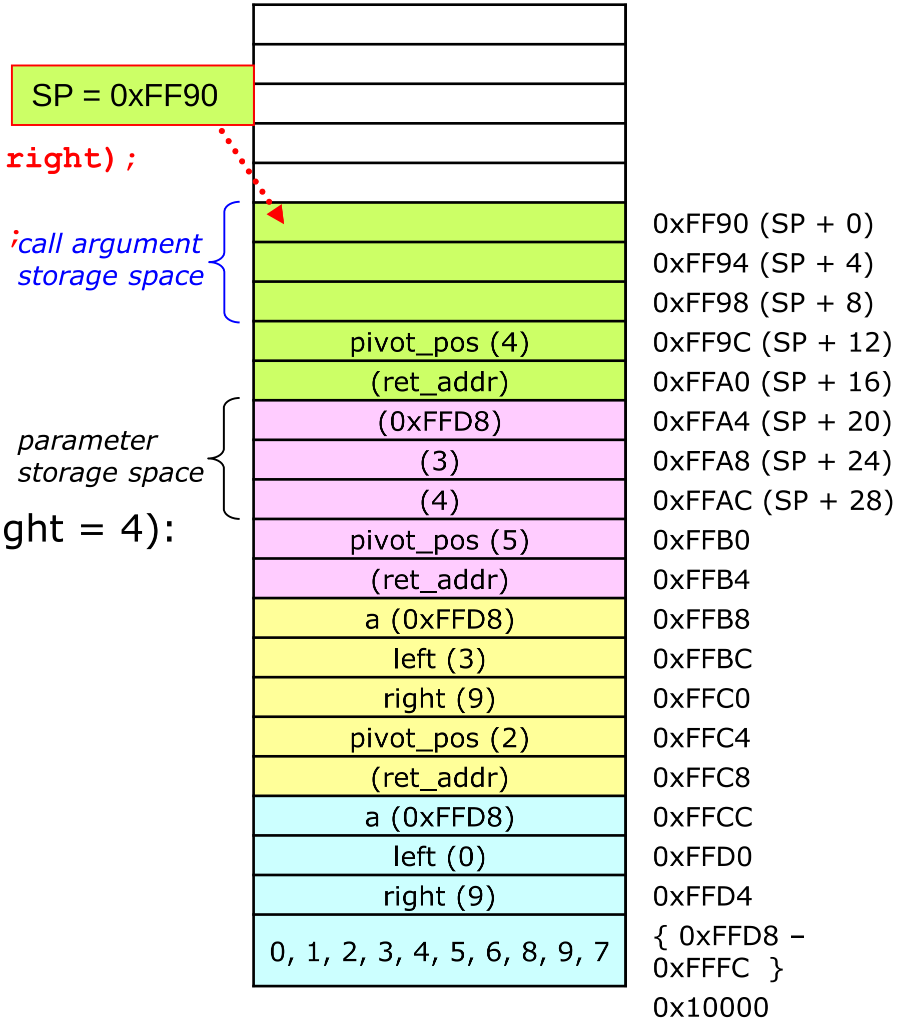




(0xFFD8)	0xFFA4 (SP + 0)
(3)	0xFFA8 (SP + 4)
(4)	0xFFAC (SP + 8)
pivot_pos (5)	0xFFB0 (SP + 12)
(ret_addr)	0xFFB4 (SP + 16)
a (0xFFD8)	0xFFB8 (SP + 20)
left (3)	0xFFBC (SP + 24)
right (9)	0xFFC0 (SP + 28)
pivot_pos (2)	0xFFC4
(ret_addr)	0xFFC8
a (0xFFD8)	0xFFCC
left (0)	0xFFD0
right (9)	0xFFD4
0, 1, 2, 3, 4, 5, 6, 8, 9, 7	{ 0xFFD8 – 0xFFFC }
	0x10000

Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```



SP = 0xFF90

call argument storage space		0xFF90 (SP + 0)
		0xFF94 (SP + 4)
		0xFF98 (SP + 8)
	pivot_pos (4)	0xFF9C (SP + 12)
	(ret_addr)	0xFFA0 (SP + 16)
parameter storage space	(0xFFD8)	0xFFA4 (SP + 20)
	(3)	0xFFA8 (SP + 24)
	(4)	0xFFAC (SP + 28)
	pivot_pos (5)	0xFFB0
	(ret_addr)	0xFFB4
	a (0xFFD8)	0xFFB8
	left (3)	0xFFBC
	right (9)	0xFFC0
	pivot_pos (2)	0xFFC4
	(ret_addr)	0xFFC8
	a (0xFFD8)	0xFFCC
	left (0)	0xFFD0
	right (9)	0xFFD4
	0, 1, 2, 3, 4, 5, 6, 8, 9, 7	{ 0xFFD8 – 0xFFFC }
		0x10000

- Enter quick_sort(a = 0xFFD8, left = 3, right = 4):
  - SP = SP – 20 = 0xFF90
  - a[3:4] = {3, 4}
- CALL partition(0xFFD8, 3, 4):

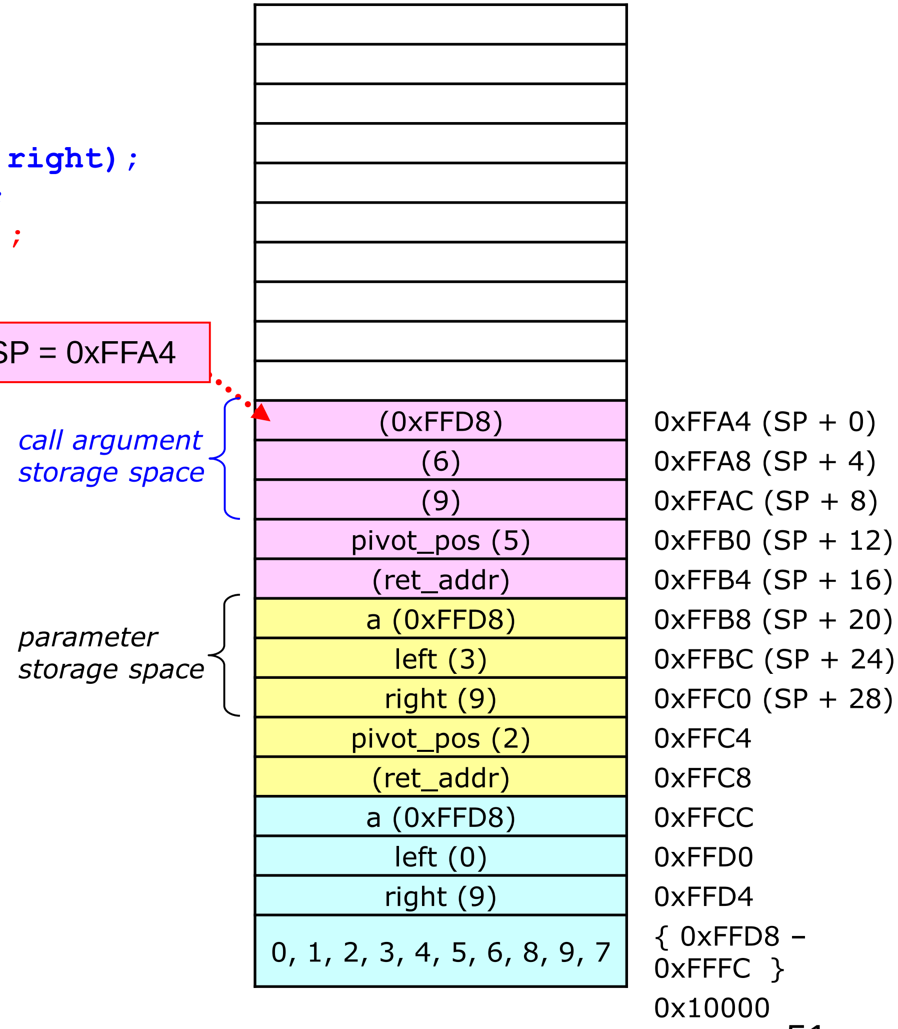
```
Output:
pivot(4): a[3:4] = {3, 4}
```

- CALL quick_sort(0xFFD8, 3, 3):
  - left = right = 3 : so recursion is skipped
- CALL quick_sort(0xFFD8, 5, 4):
  - left > right : so recursion is skipped
- Exit quick_sort:
  - SP = SP + 20 = 0xFFA4

Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
      if(left < right){
         int pivot_pos = partition(a, left, right);
         quick_sort(a, left, pivot_pos - 1);
         quick_sort(a, pivot_pos + 1, right);
      }
}
```

- CALL quick_sort(0xFFD8, 6, 9):

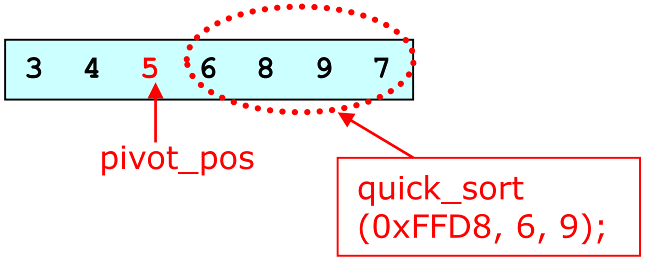


SP = 0xFFA4

call argument storage space

parameter storage space

(0xFFD8)	0xFFA4 (SP + 0)
(6)	0xFFA8 (SP + 4)
(9)	0xFFAC (SP + 8)
pivot_pos (5)	0xFFB0 (SP + 12)
(ret_addr)	0xFFB4 (SP + 16)
a (0xFFD8)	0xFFB8 (SP + 20)
left (3)	0xFFBC (SP + 24)
right (9)	0xFFC0 (SP + 28)
pivot_pos (2)	0xFFC4
(ret_addr)	0xFFC8
a (0xFFD8)	0xFFCC
left (0)	0xFFD0
right (9)	0xFFD4
0, 1, 2, 3, 4, 5, 6, 8, 9, 7	{ 0xFFD8 – 0xFFFC }
	0x10000



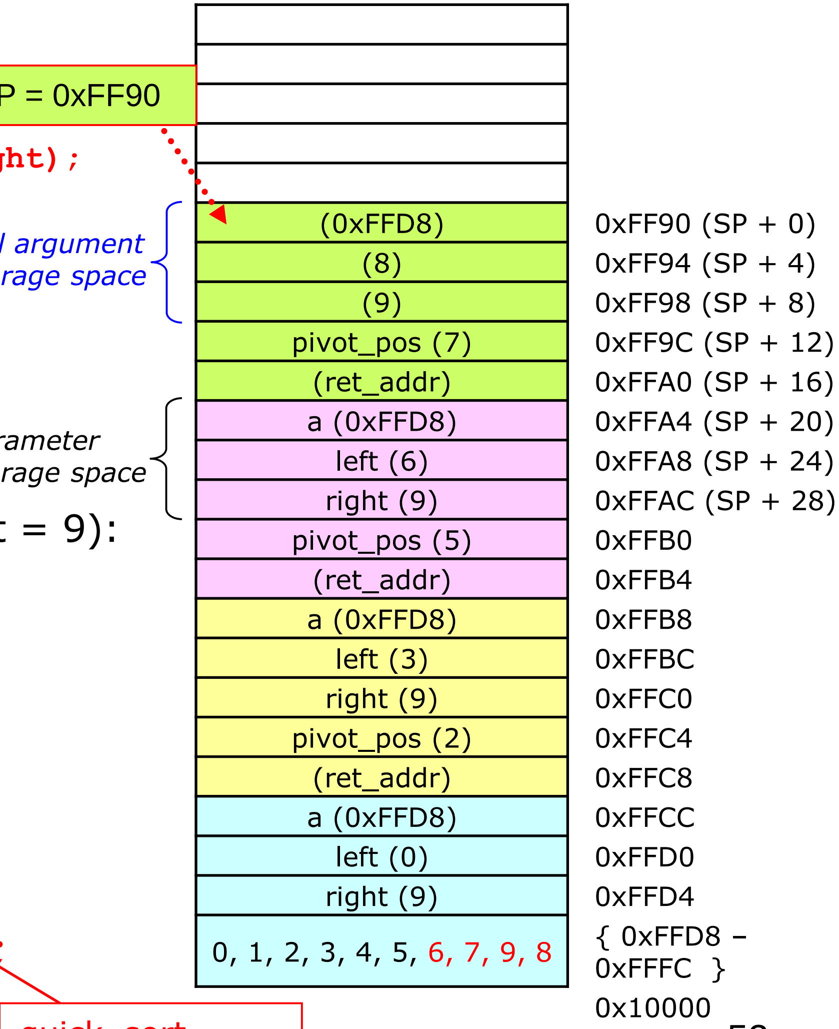
3	4	5	6	8	9	7

pivot_pos

quick_sort
(0xFFD8, 6, 9);

Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```



SP = 0xFF90

call argument storage space

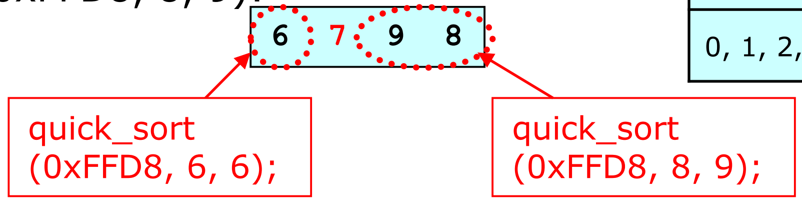
parameter storage space

(0xFFD8)	0xFF90 (SP + 0)
(8)	0xFF94 (SP + 4)
(9)	0xFF98 (SP + 8)
pivot_pos (7)	0xFF9C (SP + 12)
(ret_addr)	0xFFA0 (SP + 16)
a (0xFFD8)	0xFFA4 (SP + 20)
left (6)	0xFFA8 (SP + 24)
right (9)	0xFFAC (SP + 28)
pivot_pos (5)	0xFFB0
(ret_addr)	0xFFB4
a (0xFFD8)	0xFFB8
left (3)	0xFFBC
right (9)	0xFFC0
pivot_pos (2)	0xFFC4
(ret_addr)	0xFFC8
a (0xFFD8)	0xFFCC
left (0)	0xFFD0
right (9)	0xFFD4
0, 1, 2, 3, 4, 5, 6, 7, 9, 8	{ 0xFFD8 – 0xFFFC }
	0x10000

- Enter quick_sort(a = 0xFFD8, left = 6, right = 9):
  - SP = SP – 20 = 0xFF90
  - a[6:9] = {6, 8, 9, 7}
- CALL partition(0xFFD8, 6, 9):

```
Output:
pivot(7): a[6:9] = {6, 7, 9, 8}
```

- CALL quick_sort(0xFFD8, 6, 6):
  - left = right = 6 : so recursion is skipped
- CALL quick_sort(0xFFD8, 8, 9):

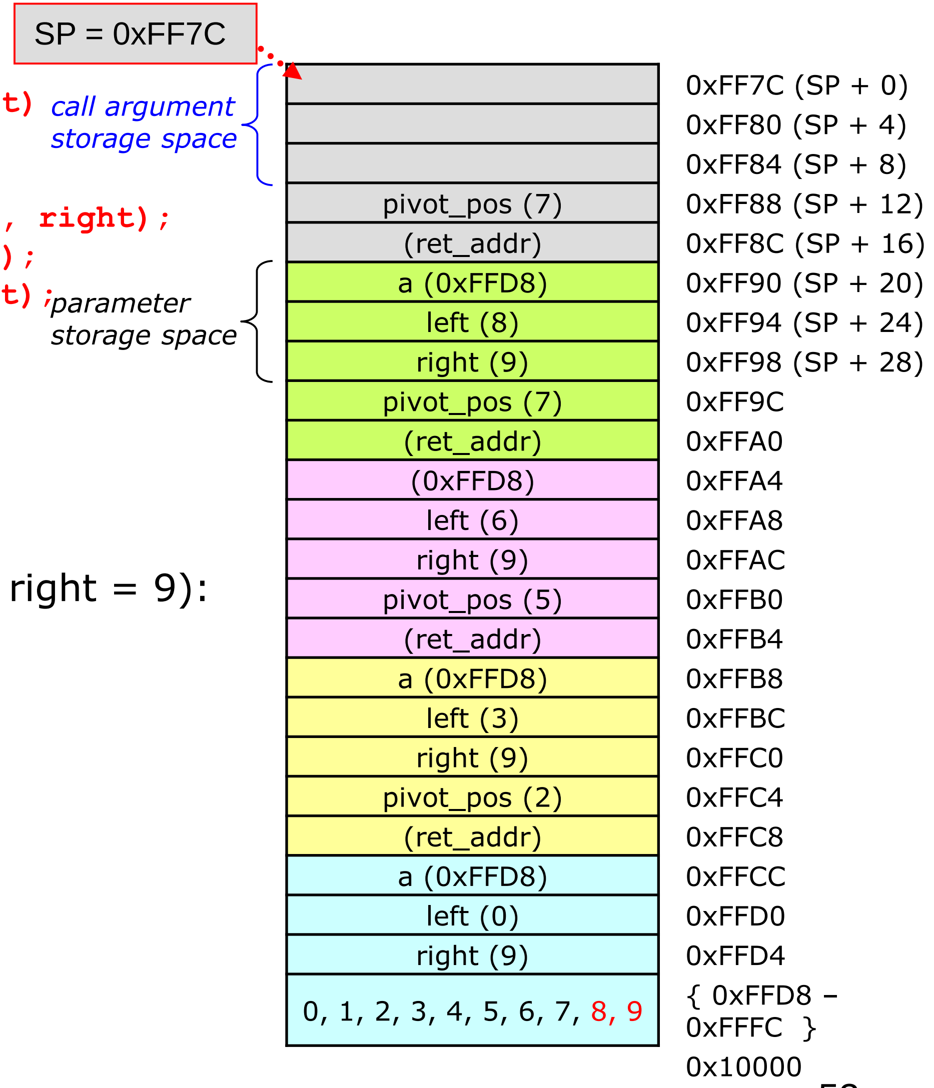


quick_sort (0xFFD8, 6, 6);

quick_sort (0xFFD8, 8, 9);

Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```



SP = 0xFF7C

call argument storage space

parameter storage space

	0xFF7C (SP + 0)
	0xFF80 (SP + 4)
	0xFF84 (SP + 8)
pivot_pos (7)	0xFF88 (SP + 12)
(ret_addr)	0xFF8C (SP + 16)
a (0xFFD8)	0xFF90 (SP + 20)
left (8)	0xFF94 (SP + 24)
right (9)	0xFF98 (SP + 28)
pivot_pos (7)	0xFF9C
(ret_addr)	0xFFA0
(0xFFD8)	0xFFA4
left (6)	0xFFA8
right (9)	0xFFAC
pivot_pos (5)	0xFFB0
(ret_addr)	0xFFB4
a (0xFFD8)	0xFFB8
left (3)	0xFFBC
right (9)	0xFFC0
pivot_pos (2)	0xFFC4
(ret_addr)	0xFFC8
a (0xFFD8)	0xFFCC
left (0)	0xFFD0
right (9)	0xFFD4
0, 1, 2, 3, 4, 5, 6, 7, 8, 9	{ 0xFFD8 – 0xFFFC }
	0x10000

- Enter quick_sort(a = 0xFFD8, left = 8, right = 9):
  - SP = SP – 20 = 0xFF7C
  - a[8:9] = {9, 8}
- CALL partition(0xFFD8, 8, 9):

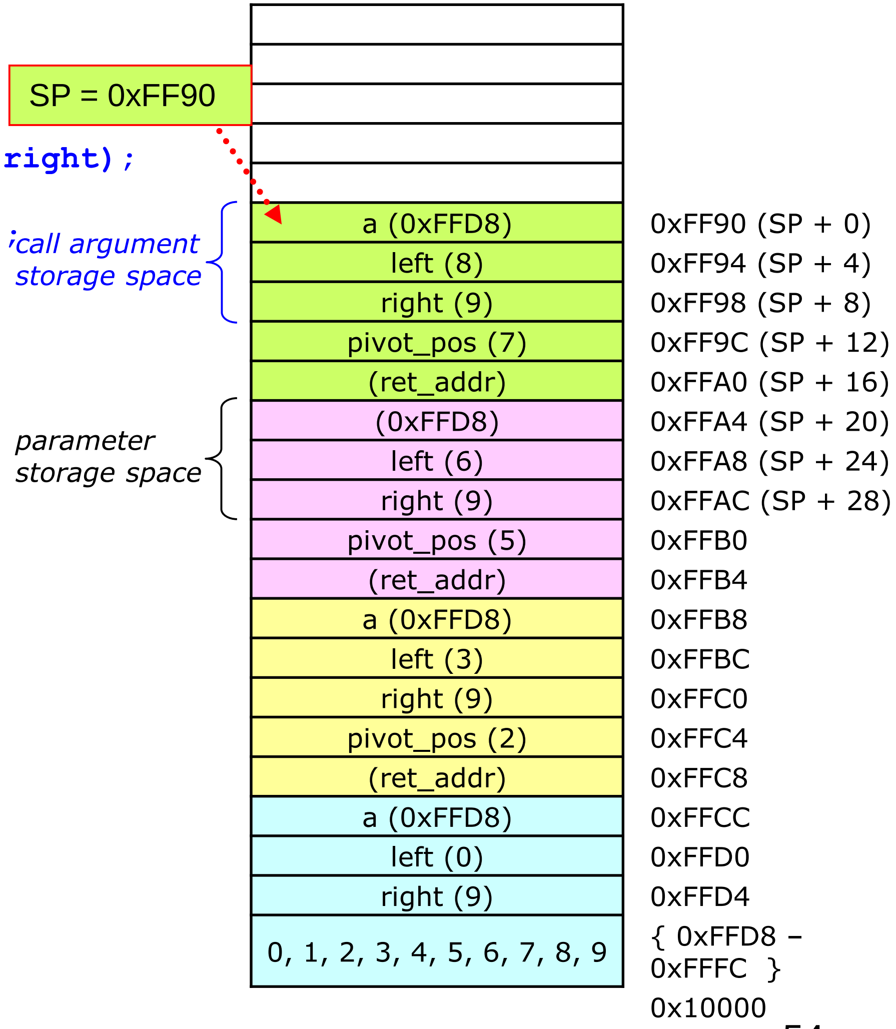
```
Output:
pivot(8): a[8:9] = {8, 9}
```

- CALL quick_sort(0xFFD8, 8, 7):
  - left > right : so recursion is skipped
- CALL quick_sort(0xFFD8, 9, 9):
  - left = right = 9 : so recursion is skipped
- Exit quick_sort:
  - SP = SP + 20 = 0xFF90

Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
        if(left < right){
            int pivot_pos = partition(a, left, right);
            quick_sort(a, left, pivot_pos - 1);
            quick_sort(a, pivot_pos + 1, right);
        }
}
```

- Exit quick_sort:
  - SP = SP + 20 = 0xFFA4



SP = 0xFF90

call argument storage space

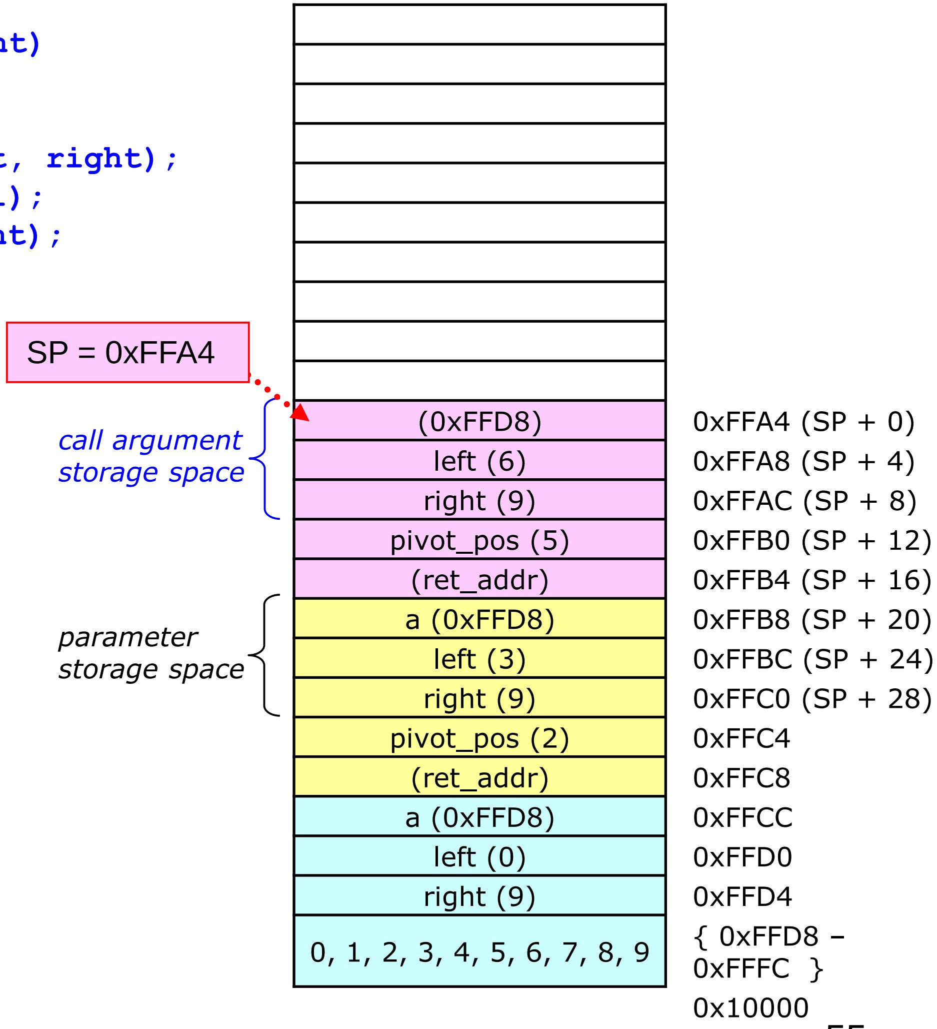
parameter storage space

a (0xFFD8)	0xFF90 (SP + 0)
left (8)	0xFF94 (SP + 4)
right (9)	0xFF98 (SP + 8)
pivot_pos (7)	0xFF9C (SP + 12)
(ret_addr)	0xFFA0 (SP + 16)
(0xFFD8)	0xFFA4 (SP + 20)
left (6)	0xFFA8 (SP + 24)
right (9)	0xFFAC (SP + 28)
pivot_pos (5)	0xFFB0
(ret_addr)	0xFFB4
a (0xFFD8)	0xFFB8
left (3)	0xFFBC
right (9)	0xFFC0
pivot_pos (2)	0xFFC4
(ret_addr)	0xFFC8
a (0xFFD8)	0xFFCC
left (0)	0xFFD0
right (9)	0xFFD4
0, 1, 2, 3, 4, 5, 6, 7, 8, 9	{ 0xFFD8 – 0xFFFC }
	0x10000

Stack Frame “quick_sort”

```
void quick_sort(int a[], int left, int right)
{
      if(left < right){
         int pivot_pos = partition(a, left, right);
         quick_sort(a, left, pivot_pos - 1);
         quick_sort(a, pivot_pos + 1, right);
      }
}
```

- Exit quick_sort:
  - SP = SP + 20 = 0xFFB8



SP = 0xFFA4

call argument storage space

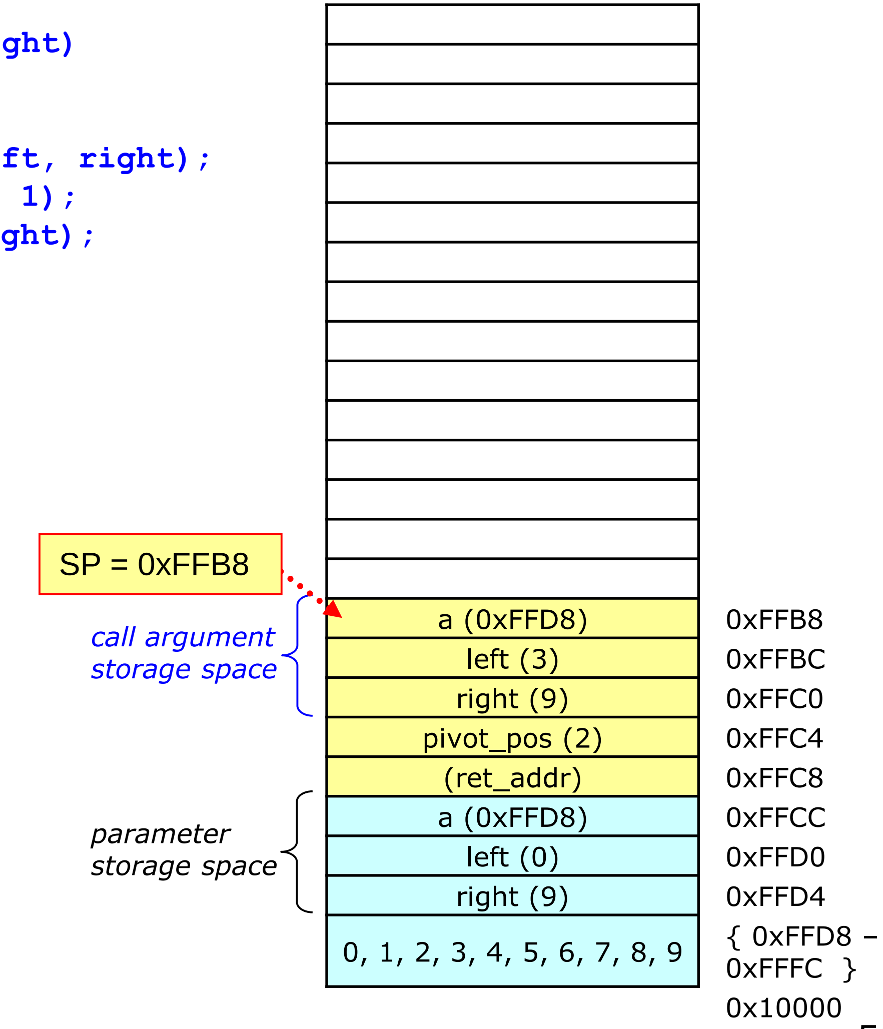
parameter storage space

(0xFFD8)	0xFFA4 (SP + 0)
left (6)	0xFFA8 (SP + 4)
right (9)	0xFFAC (SP + 8)
pivot_pos (5)	0xFFB0 (SP + 12)
(ret_addr)	0xFFB4 (SP + 16)
a (0xFFD8)	0xFFB8 (SP + 20)
left (3)	0xFFBC (SP + 24)
right (9)	0xFFC0 (SP + 28)
pivot_pos (2)	0xFFC4
(ret_addr)	0xFFC8
a (0xFFD8)	0xFFCC
left (0)	0xFFD0
right (9)	0xFFD4
0, 1, 2, 3, 4, 5, 6, 7, 8, 9	{ 0xFFD8 – 0xFFFC }
	0x10000

Stack Frame "quick_sort"

```
void quick_sort(int a[], int left, int right)
{
      if(left < right){
         int pivot_pos = partition(a, left, right);
         quick_sort(a, left, pivot_pos - 1);
         quick_sort(a, pivot_pos + 1, right);
      }
}
```

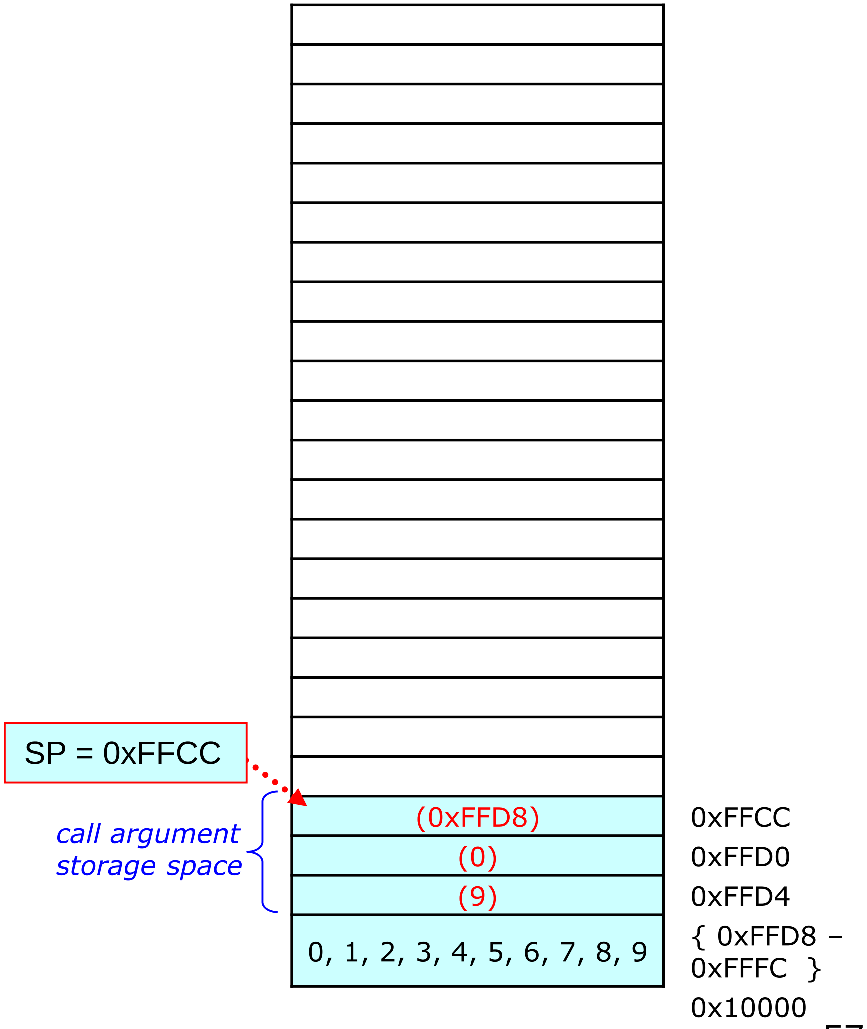
- Exit quick_sort:
  - SP = SP + 20 = 0xFFCC



Stack Frame “main”

```
#define N 10
#define RSEED 3
void main()
{
        int a[N]; /* size = 4N = 40 bytes */
        create_array(a, N, RSEED);
        print_array2(a, 0, N - 1);
        quick_sort(a, 0, N – 1);
        print_array2(a, 0, N - 1);
}
```

- CALL partition(0xFFD8, 0, 9):
- Exit main:
  - SP = SP + 52 = 0x10000



SP = 0xFFCC

call argument storage space

(0xFFD8)	0xFFCC
(0)	0xFFD0
(9)	0xFFD4
0, 1, 2, 3, 4, 5, 6, 7, 8, 9	{ 0xFFD8 – 0xFFFC }
	0x10000

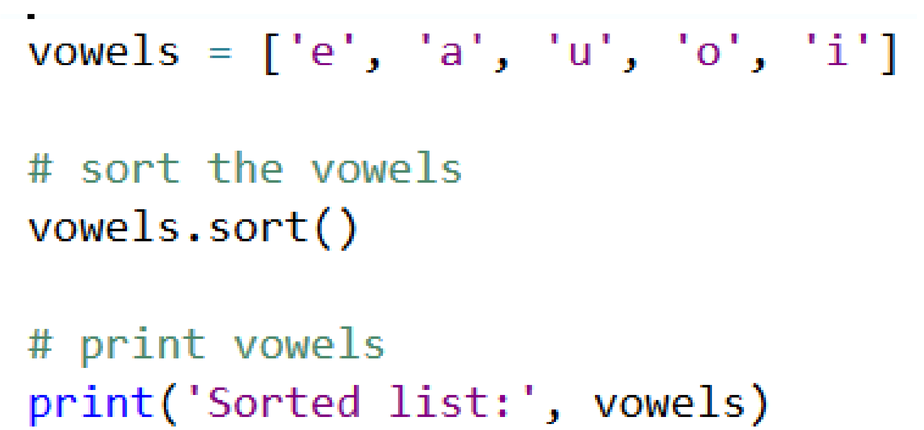
Python Programming

Python List sort()

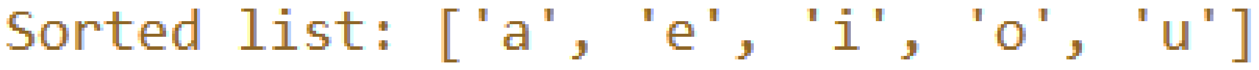
The sort() method sorts the elements of a given list in a specific order - Ascending or Descending.

The syntax of sort() method is:

list.sort(key=..., reverse=...)

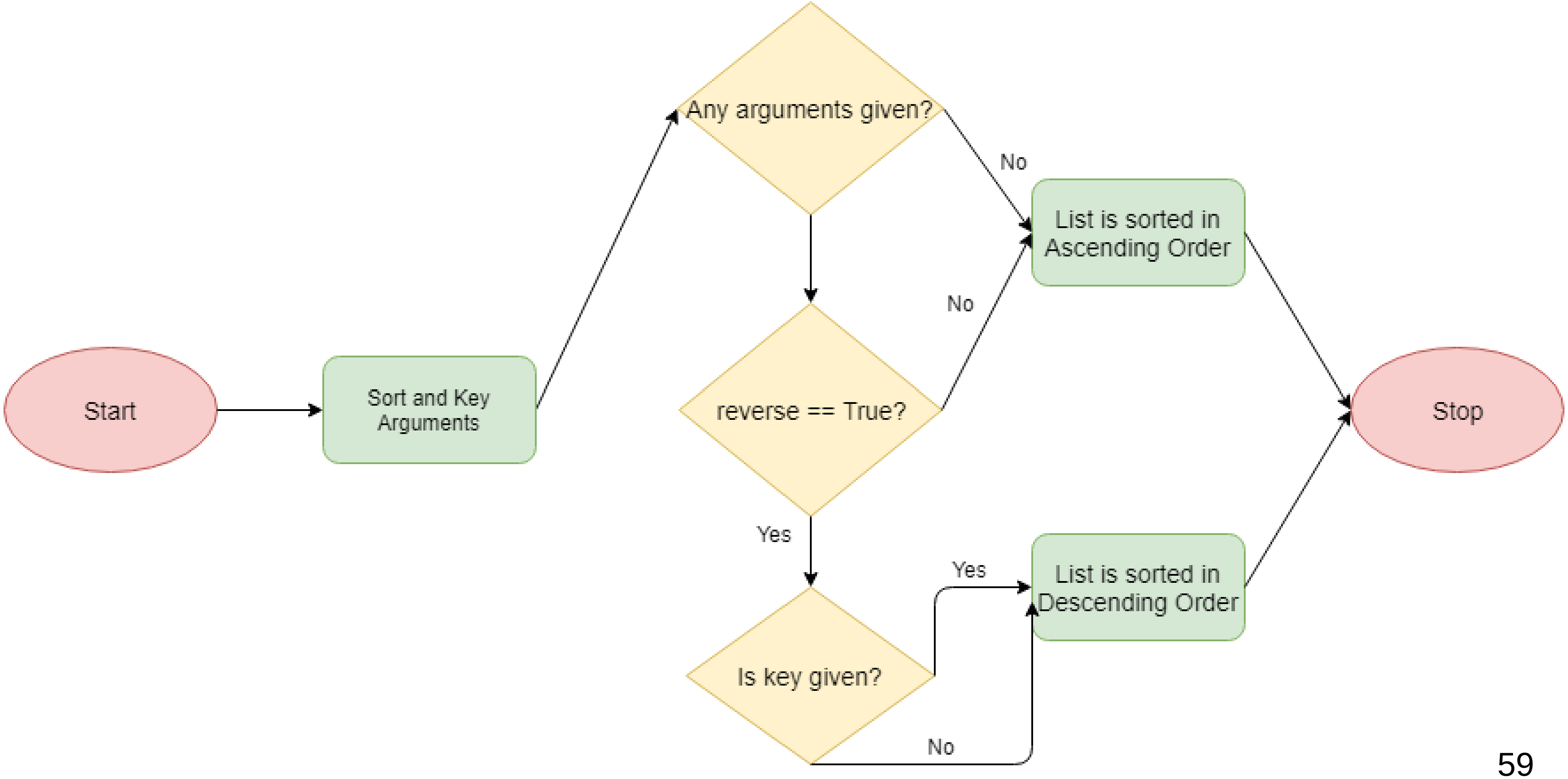


```
vowels = ['e', 'a', 'u', 'o', 'i']

# sort the vowels
vowels.sort()

# print vowels
print('Sorted list:', vowels)
```

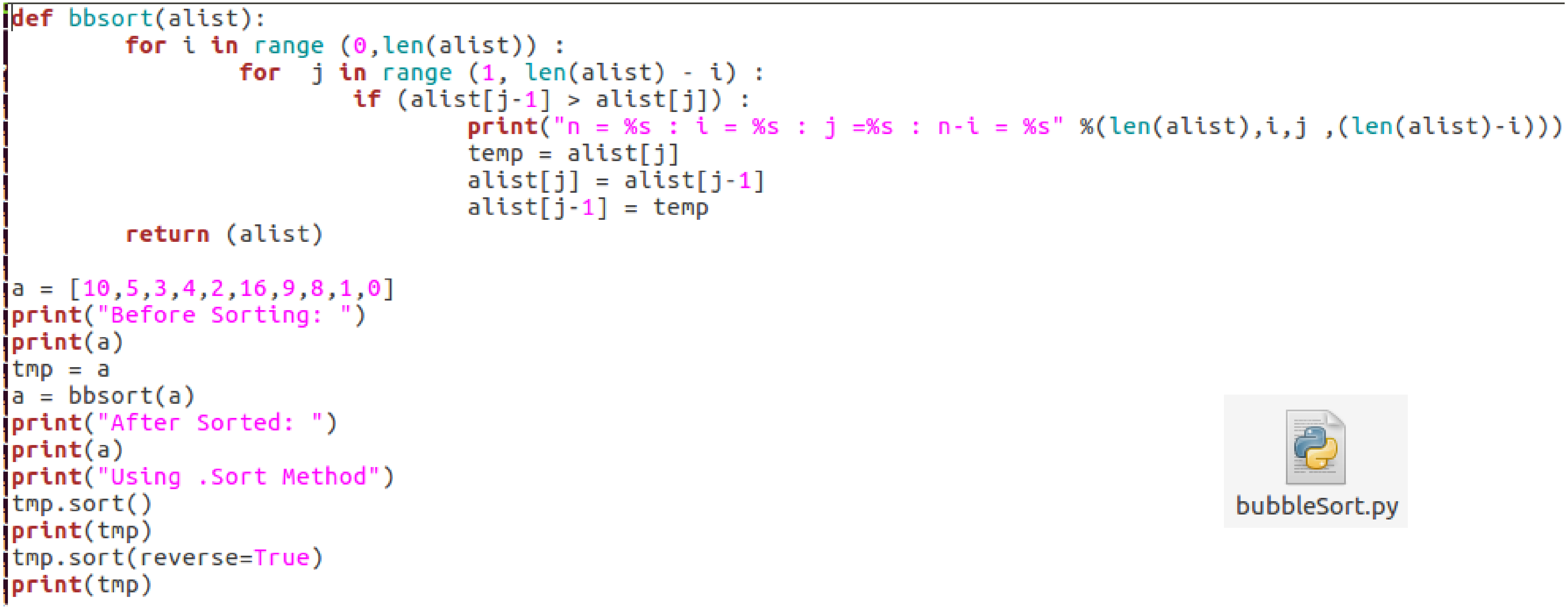


```
Sorted list: ['a', 'e', 'i', 'o', 'u']
```

Python uses an algorithm called Timsort:
Timsort is a hybrid sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It was invented by Tim Peters in 2002 for use in the Python programming language.



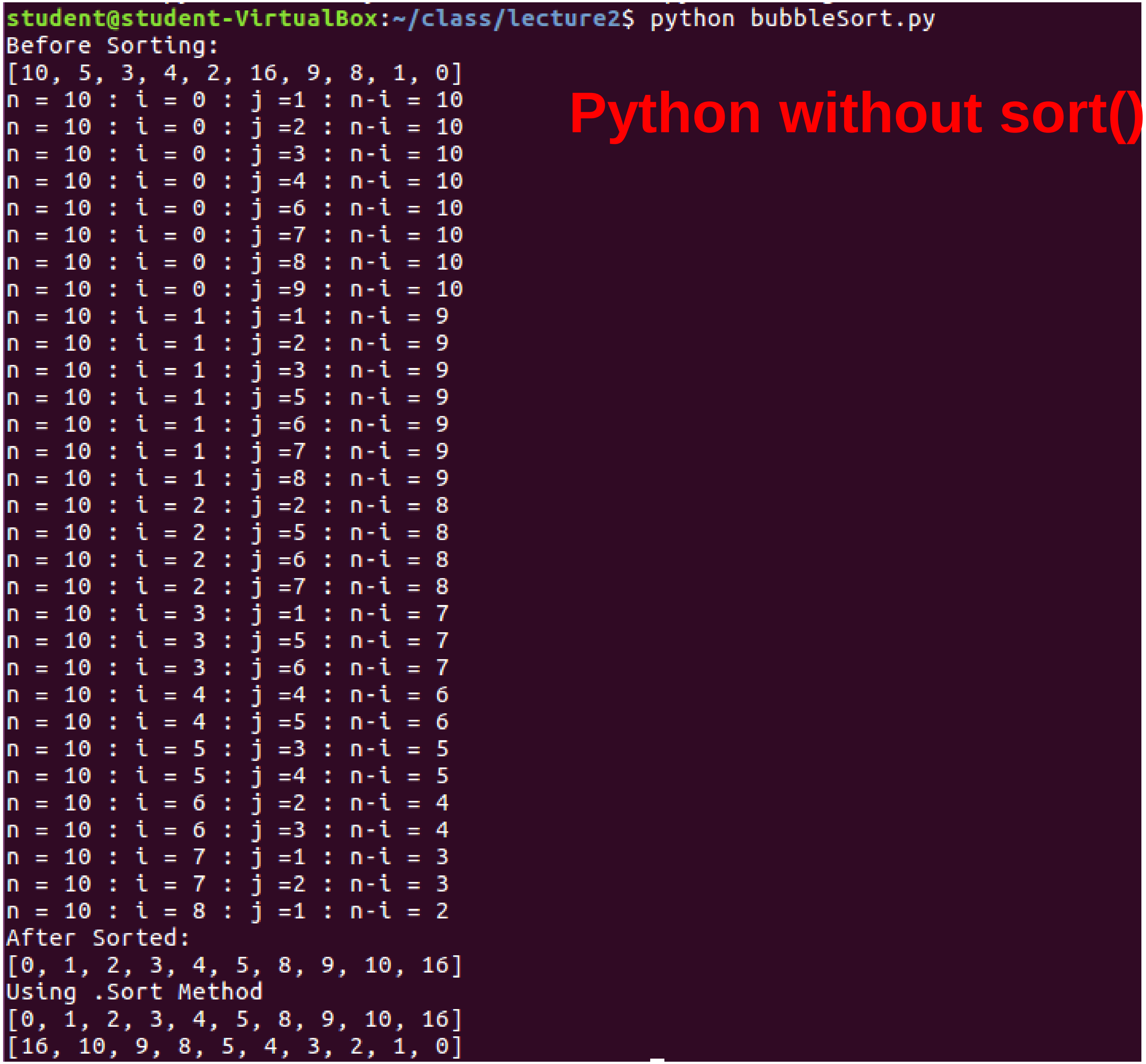
Python bubble sort without sort() method



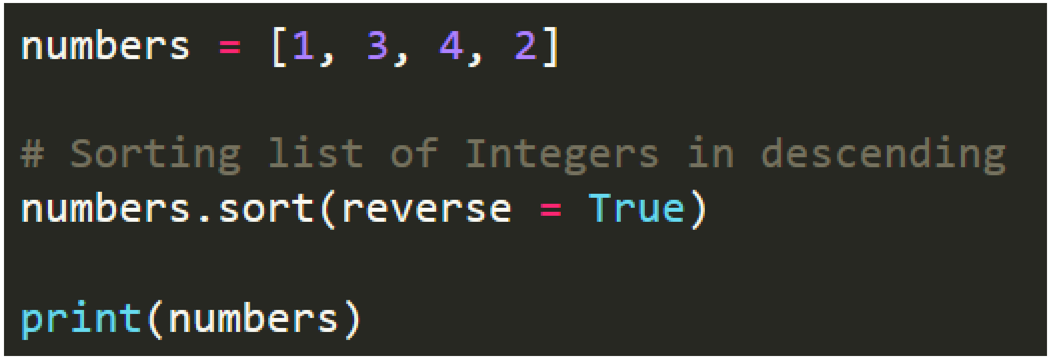
```python
def bbsort(alist):
        for i in range (0,len(alist)) :
                for  j in range (1, len(alist) - i) :
                        if (alist[j-1] > alist[j]) :
                                print("n = %s : i = %s : j =%s : n-i = %s" %(len(alist),i,j ,(len(alist)-i)))
                                temp = alist[j]
                                alist[j] = alist[j-1]
                                alist[j-1] = temp
        return (alist)

a = [10,5,3,4,2,16,9,8,1,0]
print("Before Sorting: ")
print(a)
tmp = a
a = bbsort(a)
print("After Sorted: ")
print(a)
print("Using .Sort Method")
tmp.sort()
print(tmp)
tmp.sort(reverse=True)
print(tmp)
```

bubbleSort.py

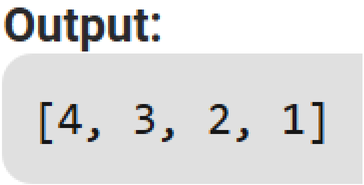
Python without sort() method



```
student@student-VirtualBox:~/class/lecture2$ python bubbleSort.py
Before Sorting:
[10, 5, 3, 4, 2, 16, 9, 8, 1, 0]
n = 10 : i = 0 : j =1 : n-i = 10
n = 10 : i = 0 : j =2 : n-i = 10
n = 10 : i = 0 : j =3 : n-i = 10
n = 10 : i = 0 : j =4 : n-i = 10
n = 10 : i = 0 : j =6 : n-i = 10
n = 10 : i = 0 : j =7 : n-i = 10
n = 10 : i = 0 : j =8 : n-i = 10
n = 10 : i = 0 : j =9 : n-i = 10
n = 10 : i = 1 : j =1 : n-i = 9
n = 10 : i = 1 : j =2 : n-i = 9
n = 10 : i = 1 : j =3 : n-i = 9
n = 10 : i = 1 : j =5 : n-i = 9
n = 10 : i = 1 : j =6 : n-i = 9
n = 10 : i = 1 : j =7 : n-i = 9
n = 10 : i = 1 : j =8 : n-i = 9
n = 10 : i = 2 : j =2 : n-i = 8
n = 10 : i = 2 : j =5 : n-i = 8
n = 10 : i = 2 : j =6 : n-i = 8
n = 10 : i = 2 : j =7 : n-i = 8
n = 10 : i = 3 : j =1 : n-i = 7
n = 10 : i = 3 : j =5 : n-i = 7
n = 10 : i = 3 : j =6 : n-i = 7
n = 10 : i = 4 : j =4 : n-i = 6
n = 10 : i = 4 : j =5 : n-i = 6
n = 10 : i = 5 : j =3 : n-i = 5
n = 10 : i = 5 : j =4 : n-i = 5
n = 10 : i = 6 : j =2 : n-i = 4
n = 10 : i = 6 : j =3 : n-i = 4
n = 10 : i = 7 : j =1 : n-i = 3
n = 10 : i = 7 : j =2 : n-i = 3
n = 10 : i = 8 : j =1 : n-i = 2
After Sorted:
[0, 1, 2, 3, 4, 5, 8, 9, 10, 16]
Using .Sort Method
[0, 1, 2, 3, 4, 5, 8, 9, 10, 16]
[16, 10, 9, 8, 5, 4, 3, 2, 1, 0]
```

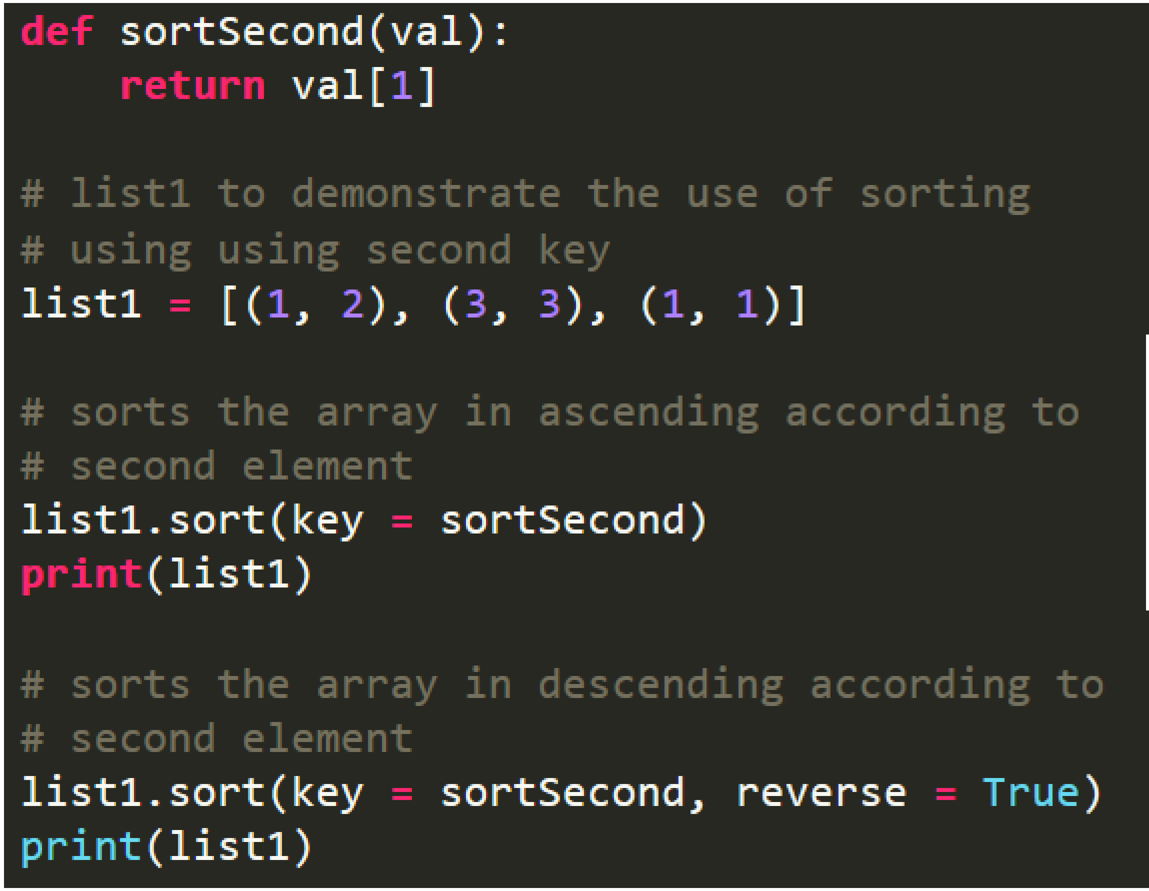


```python
numbers = [1, 3, 4, 2]

# Sorting list of Integers in descending
numbers.sort(reverse = True)

print(numbers)
```

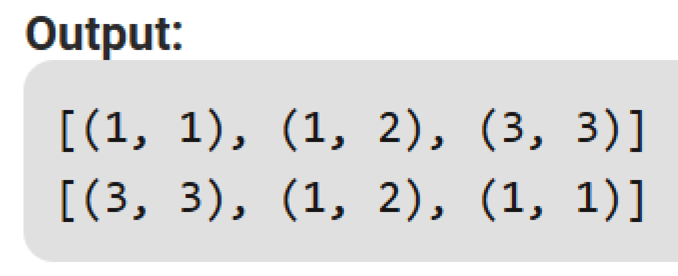


Output:

```
[4, 3, 2, 1]
```



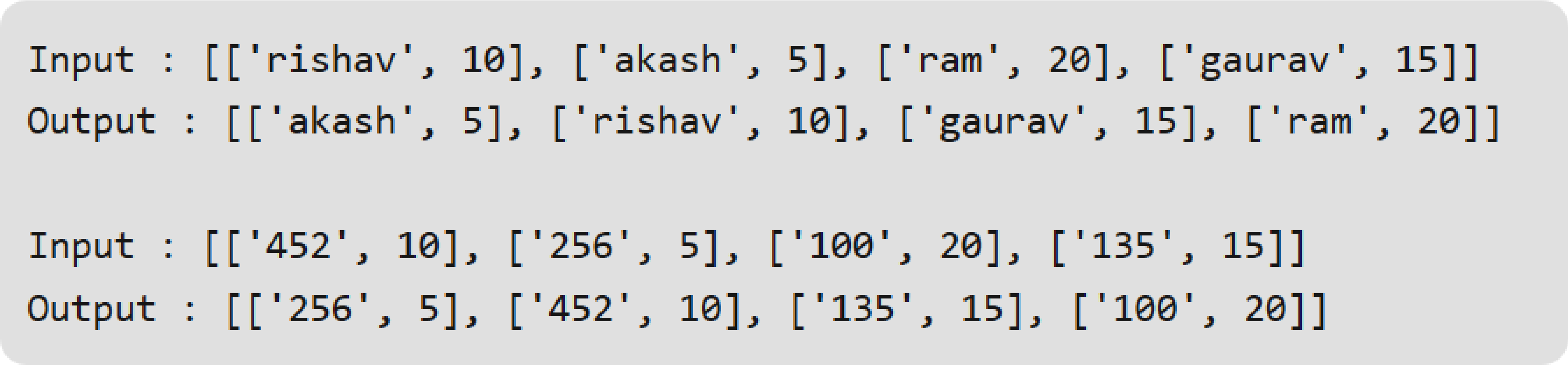
```python
def sortSecond(val):
    return val[1]

# list1 to demonstrate the use of sorting
# using using second key
list1 = [(1, 2), (3, 3), (1, 1)]

# sorts the array in ascending according to
# second element
list1.sort(key = sortSecond)
print(list1)

# sorts the array in descending according to
# second element
list1.sort(key = sortSecond, reverse = True)
print(list1)
```



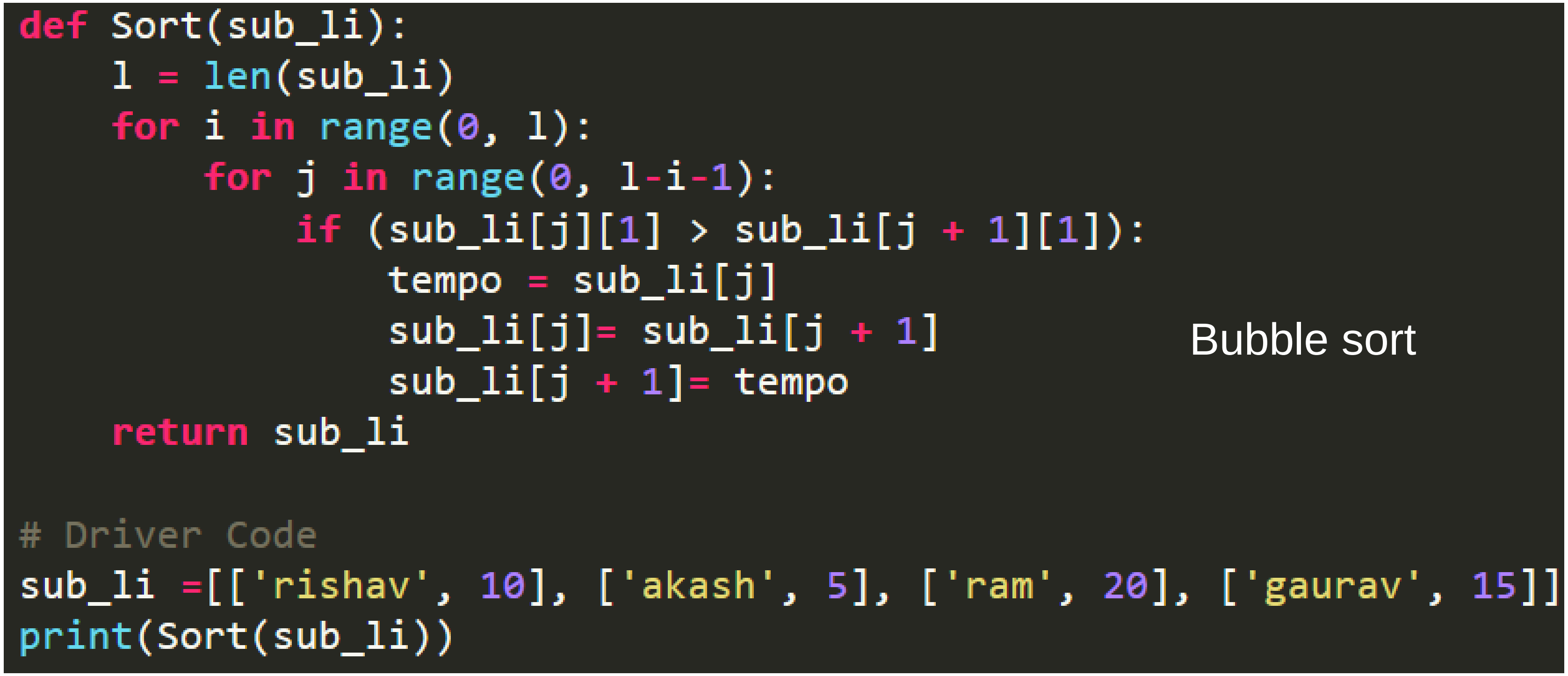
Output:

```
[(1, 1), (1, 2), (3, 3)]
[(3, 3), (1, 2), (1, 1)]
```

Python | Sort a list according to the second element in sublist



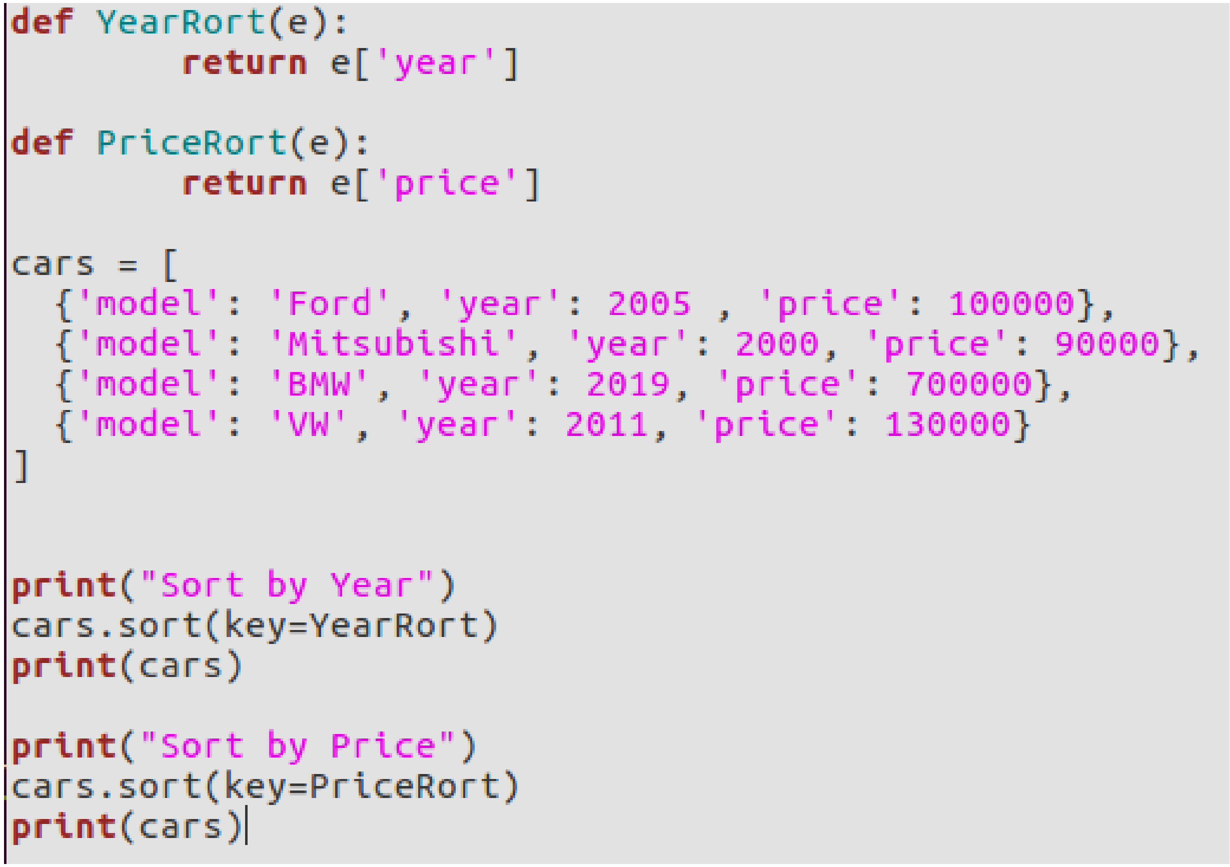
```
Input : [['rishav', 10], ['akash', 5], ['ram', 20], ['gaurav', 15]]
Output : [['akash', 5], ['rishav', 10], ['gaurav', 15], ['ram', 20]]

Input : [['452', 10], ['256', 5], ['100', 20], ['135', 15]]
Output : [['256', 5], ['452', 10], ['135', 15], ['100', 20]]
```



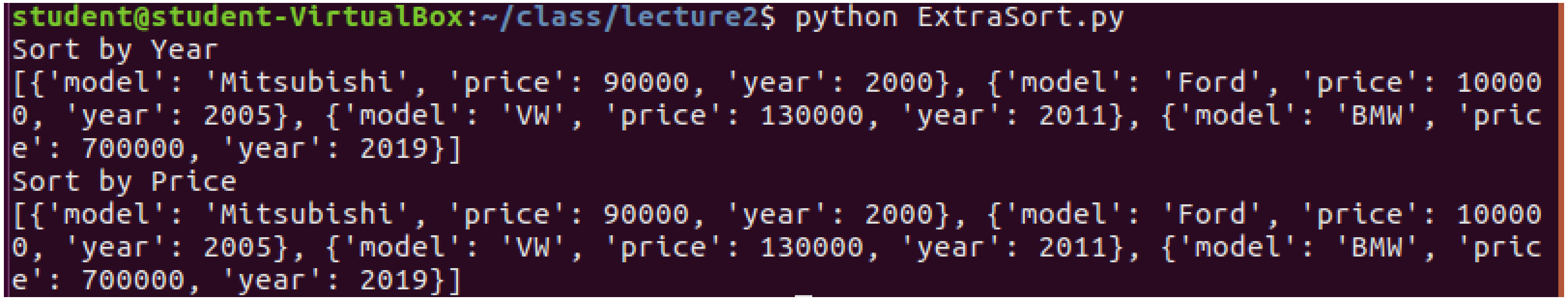
```python
def Sort(sub_li):
    l = len(sub_li)
    for i in range(0, l):
        for j in range(0, l-i-1):
            if (sub_li[j][1] > sub_li[j + 1][1]):
                tempo = sub_li[j]
                sub_li[j]= sub_li[j + 1]
                sub_li[j + 1]= tempo
    return sub_li

# Driver Code
sub_li =[['rishav', 10], ['akash', 5], ['ram', 20], ['gaurav', 15]]
print(Sort(sub_li))
```

Bubble sort

Python | Sort a list according to the any element in sublist

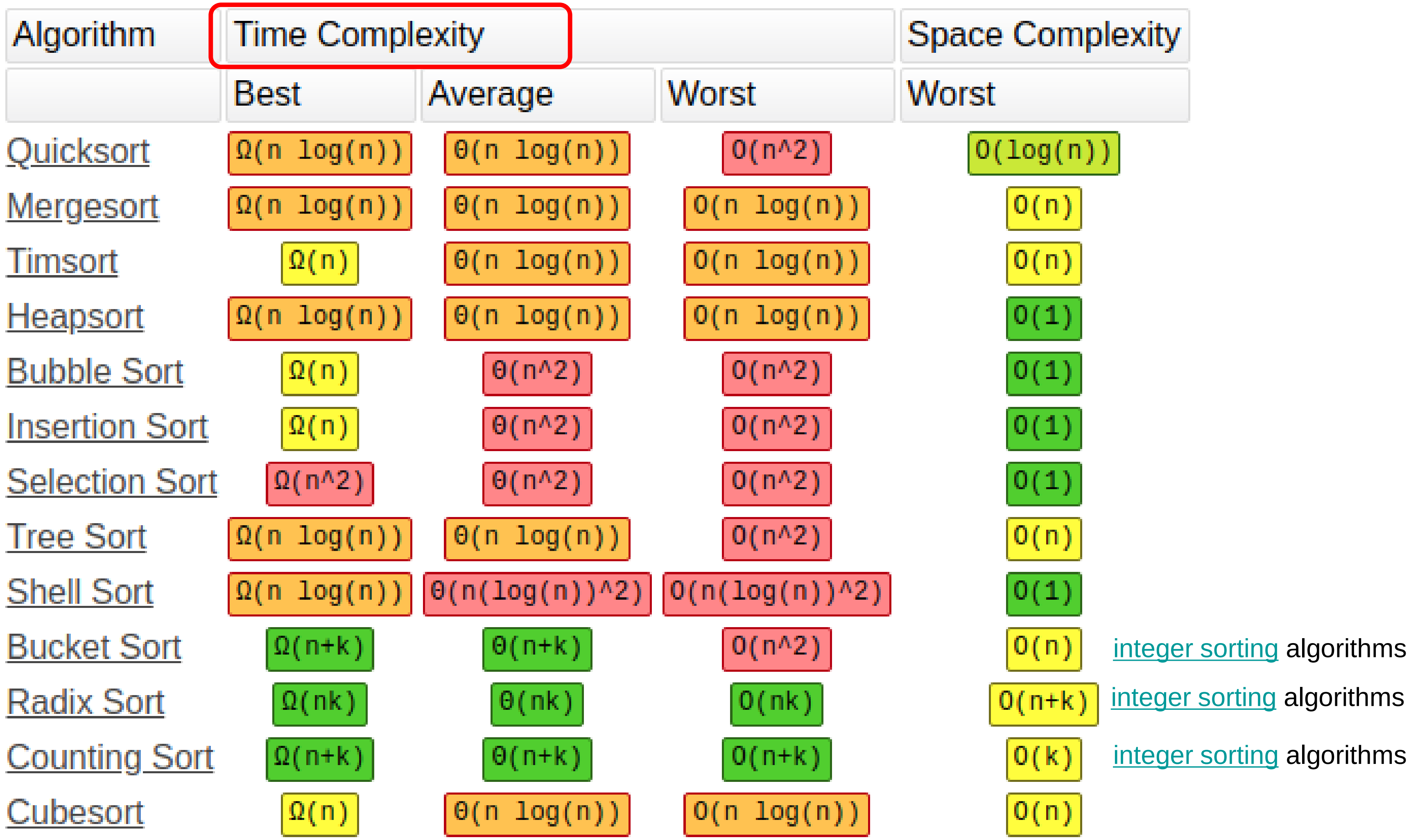


```
def YearRort(e):
        return e['year']

def PriceRort(e):
        return e['price']

cars = [
  {'model': 'Ford', 'year': 2005 , 'price': 100000},
  {'model': 'Mitsubishi', 'year': 2000, 'price': 90000},
  {'model': 'BMW', 'year': 2019, 'price': 700000},
  {'model': 'VW', 'year': 2011, 'price': 130000}
]


print("Sort by Year")
cars.sort(key=YearRort)
print(cars)

print("Sort by Price")
cars.sort(key=PriceRort)
print(cars)
```



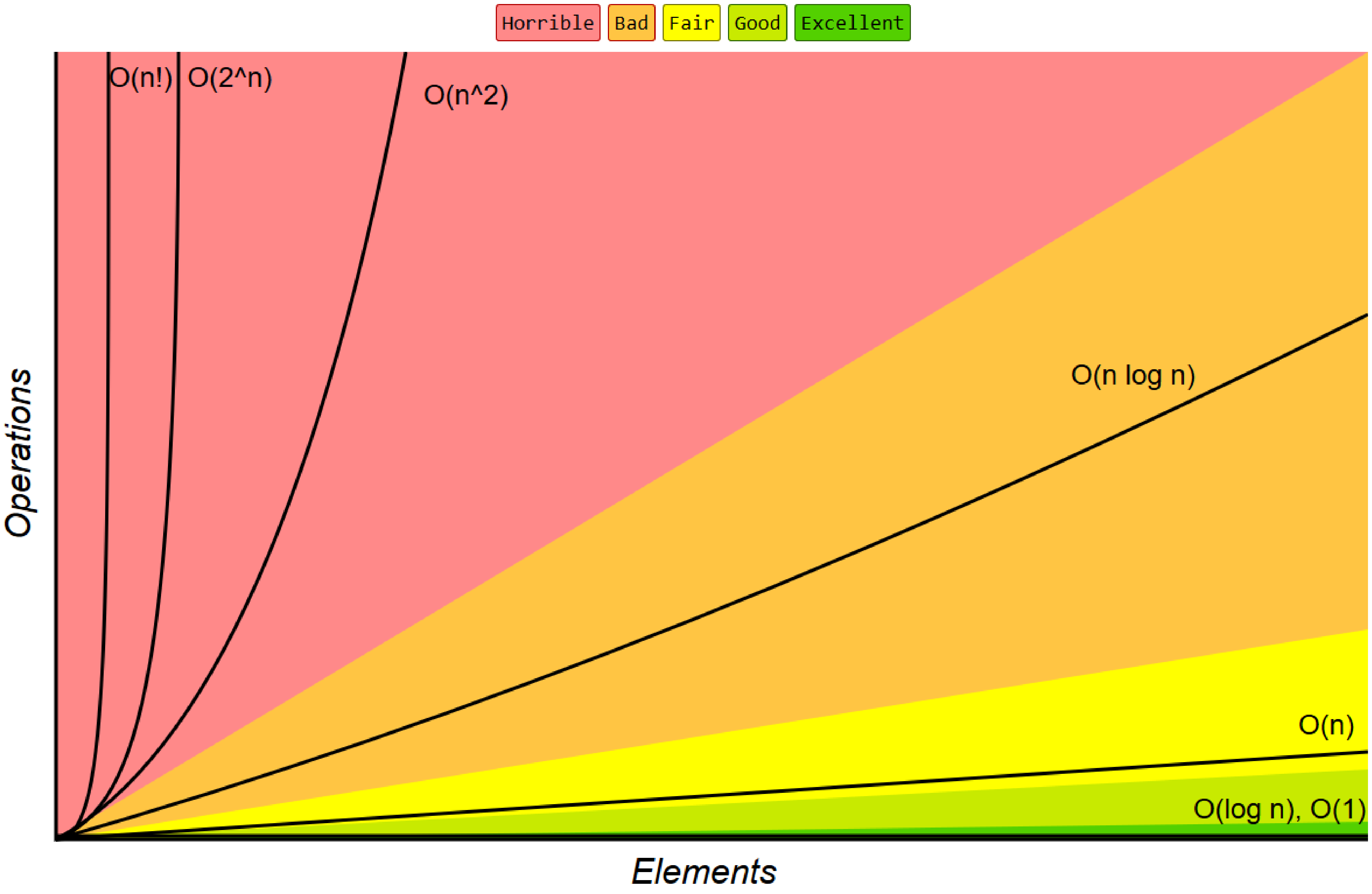
```
student@student-VirtualBox:~/class/lecture2$ python ExtraSort.py
Sort by Year
[{'model': 'Mitsubishi', 'price': 90000, 'year': 2000}, {'model': 'Ford', 'price': 10000
0, 'year': 2005}, {'model': 'VW', 'price': 130000, 'year': 2011}, {'model': 'BMW', 'pric
e': 700000, 'year': 2019}]
Sort by Price
[{'model': 'Mitsubishi', 'price': 90000, 'year': 2000}, {'model': 'Ford', 'price': 10000
0, 'year': 2005}, {'model': 'VW', 'price': 130000, 'year': 2011}, {'model': 'BMW', 'pric
e': 700000, 'year': 2019}]
```

Array Sorting Algorithms



Algorithm	Time Complexity			Space Complexity	
	Best	Average	Worst	Worst	
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))	
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)	
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)	
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)	
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)	
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)	
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)	
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)	
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)	
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)	integer sorting algorithms
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)	integer sorting algorithms
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)	integer sorting algorithms
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)	

Big-O Complexity Chart

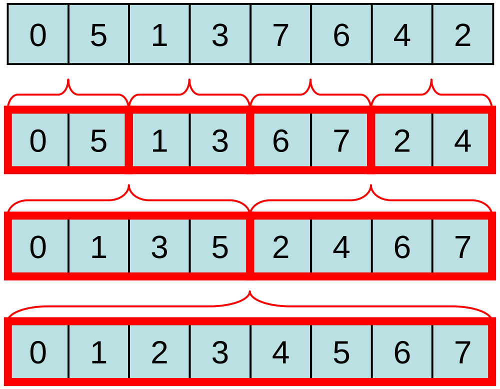


Exercise 2 (sorting)

1. Write a program that sorts words in dictionary order (words should be given from the command arguments)
   - “this” “is” “a” “pen” → “a” “is” “pen” “this”
2. Write a program that prints the “median” value
   - A median is the element which is in the middle of the sorted list, so you can compute the median by sorting the list and printing the middle element
   - Think about how you can get the median value without sorting the entire list by modifying the quick_sort program

Exercise 2 (sorting)

3. Write a sorting program using “merge sort” algorithm on $M = 2^N$ array elements:
   - For $k = 0, 1, \ldots, N - 1$: Do sorting on every 2^{k+1} adjacent elements in the array
     - At $k = 0$, sorting each 2 adjacent elements requires one comparison and a swap (if order is reversed)
     - At $k > 0$, on each 2^{k+1} adjacent elements, the first 2^k elements and the second 2^k elements are already sorted by the previous iteration



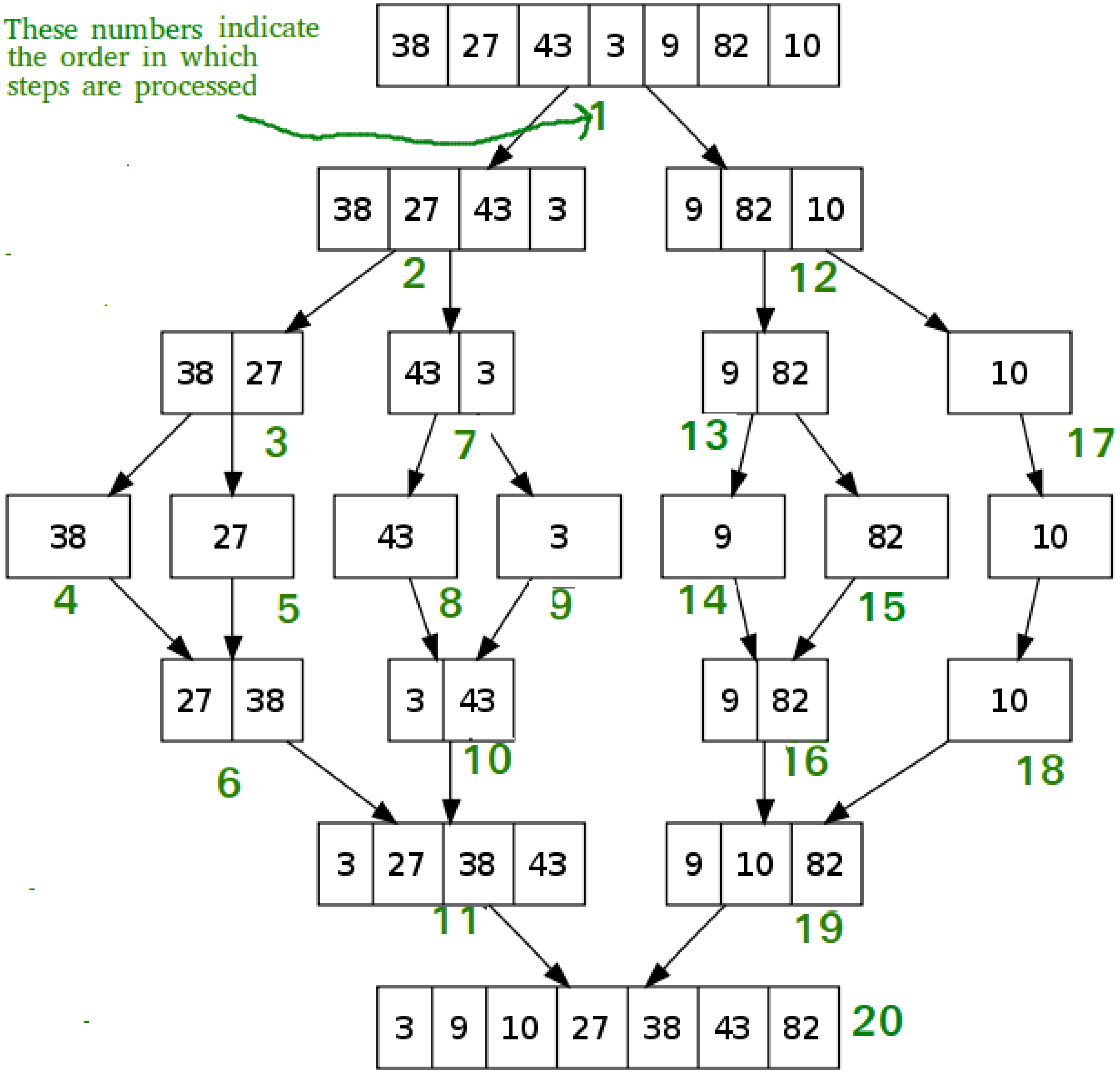
0 5 1 3 7 6 4 2

0 5 1 3 6 7 2 4

0 1 3 5 2 4 6 7

0 1 2 3 4 5 6 7

4. Modify your above program so that it can also work on the array size which is not a power of 2



These numbers indicate the order in which steps are processed

Merge Sort