Lecture 11. Quicksort

- **Sort `x[0..n-1]` into increasing (or decreasing) order**
- **Quicksort is a well-known sorting algorithm: Recursion is natural and fast**

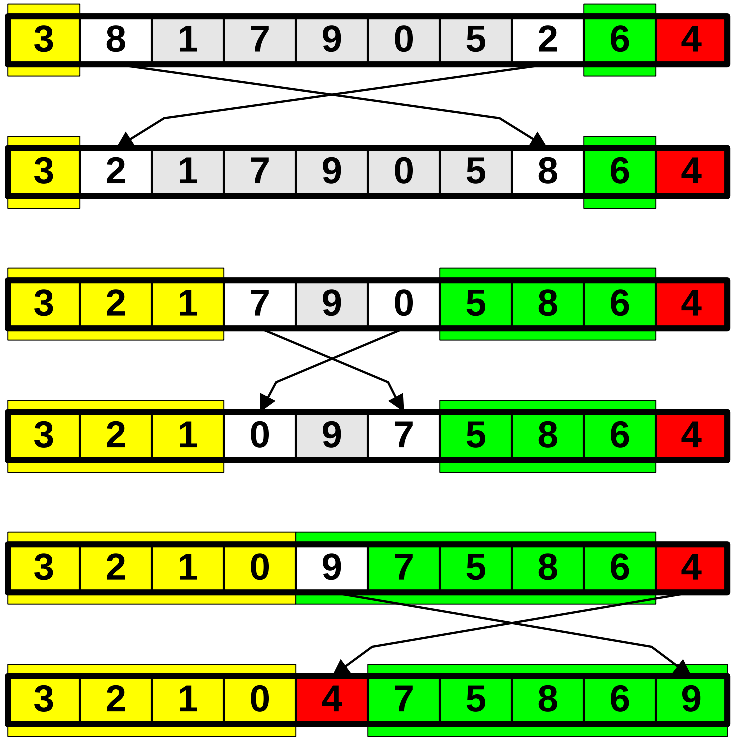
To sort `x[0..n-1]`:

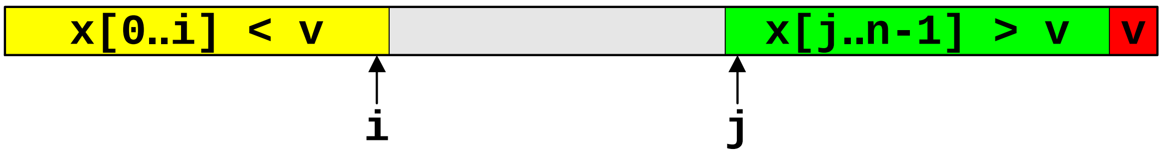
1. **Pick a ‘pivot’ element**
2. **Rearrange x so that:**
   `x[k]` holds this element, `x[0..k-1]` $<$ `x[k]`, and `x[k+1..n-1]` $>$ `x[k]`
3. **Sort `x[0..k-1]` and `x[k+1..n-1]` recursively**

```
void quicksort(int x[], int l, int r) {
    if (r > l) {
        int k = partition(x, l, r);
        quicksort(x, l, k - 1);
        quicksort(x, k + 1, r);
    }
}

int main(void) {
    int n, array[1000];

    ...
    quicksort(array, 0, n - 1);
    ...
}
```

Partitioning

```
int partition(int x[], int i, int j) {
    int k = j, v = x[k];

    i--;
    while (i < j) {
        while (          x[++i] < v)
            ;
        while (--j > i && x[  j] > v)
            ;
        if (i < j) {
            int t = x[i];
            x[i] = x[j];
            x[j] = t;
        }
    }
    x[k] = x[i];
    x[i] = v;
    return i;
}
```



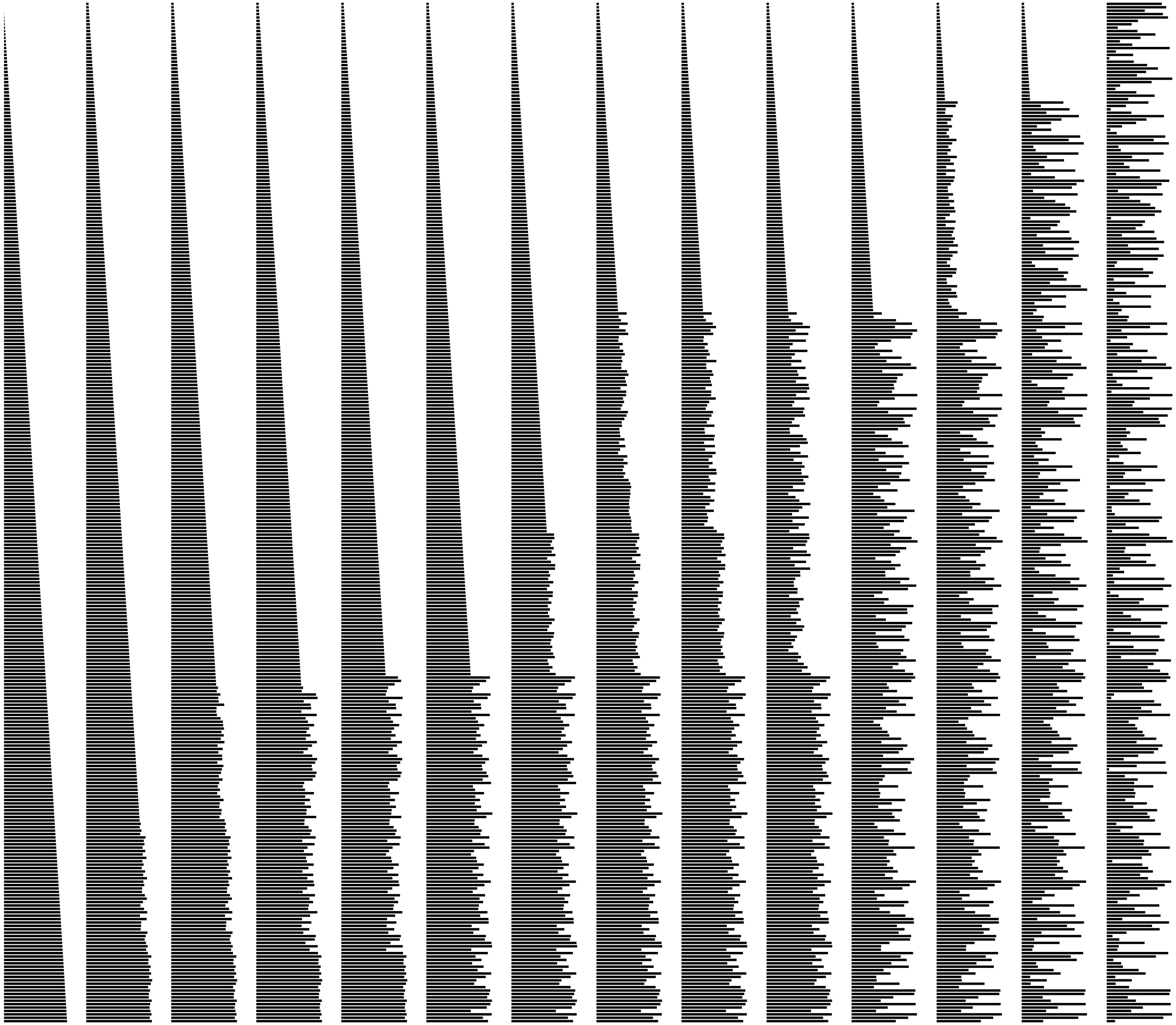


- **For more, read R. Sedgewick, *Algorithms in C*, Addison-Wesley, 1990**

Quicksort in Action

`quicksort(x, 0, 9)`	3	8	1	7	9	0	5	2	6	4
	3	*2*	1	7	9	0	5	*8*	6	4
	3	2	1	*0*	9	*7*	5	8	6	4
	3	2	1	0	*4*	7	5	8	6	*9*
`quicksort(x, 0, 3)`	3	2	1	0	4	7	5	8	6	9
	0	2	1	*3*	4	7	5	8	6	9
`quicksort(x, 0, -1)`										
`quicksort(x, 1, 3)`	0	2	1	3	4	7	5	8	6	9
	0	2	1	3	4	7	5	8	6	9
`quicksort(x, 1, 2)`	0	2	1	3	4	7	5	8	6	9
	0	*1*	*2*	3	4	7	5	8	6	9
`quicksort(x, 1, 0)`										
`quicksort(x, 2, 2)`										
`quicksort(x, 4, 3)`										
`quicksort(x, 5, 9)`	0	1	2	3	4	7	5	8	6	9
	0	1	2	3	4	7	5	8	6	9
`quicksort(x, 5, 8)`	0	1	2	3	4	7	5	8	6	9
	0	1	2	3	4	*5*	*7*	8	6	9
	0	1	2	3	4	5	*6*	8	*7*	9
`quicksort(x, 5, 5)`										
`quicksort(x, 7, 8)`	0	1	2	3	4	5	6	8	7	9
	0	1	2	3	4	5	6	*7*	*8*	9
`quicksort(x, 7, 6)`										
`quicksort(x, 8, 8)`										
`quicksort(x, 10, 9)`										

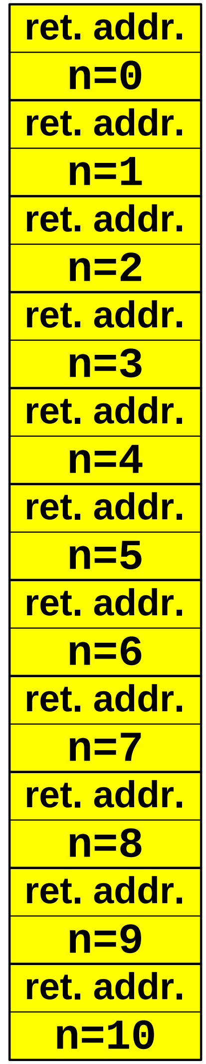
Quicksort in Action, cont'd



Implementing Recursive Functions

- **Consider `sum(10)`: *each call* must have *its own argument* n and its return address**
- **Use a *stack* to hold arguments, local variables, and the return address**

```
sum(n=10) calls
    sum(9)
        sum(8)
            sum(7)
                sum(6)
                    sum(5)
                        sum(4)
                            sum(3)
                                sum(2)
                                    sum(1)
                                        sum(0)
                                        returns 0
                                    returns 1
                                returns 3
                            returns 6
                        returns 10
                    returns 15
                returns 21
            returns 28
        returns 36
    returns 45
returns 55
```



Implementing Recursive Functions, cont'd

- Use ***conventions*** for the stack and for how arguments, etc. are 'pushed'

  Use R_7 as the 'stack pointer:' it holds the address of the top element

  Stack starts at FF_{16} and grows 'down' — toward *lower* addresses

  Push the arguments onto the stack before calling a function; push the return address upon entering a function

```
30: B201   R2 <- 1                push the return address
31: 2772   R7 <- R7 - R2 = R7 - 1
32: A670   M[R7+0] <- R6
33: 9171   R1 <- M[R7+1]          R1 <- n
34: 2312   R3 <- R1 - R2 = R1 - 1 R3 <- n - 1
35: 633D   jump to 3D if R3 < 0   if (n == 0) return 0
36: 2772   R7 <- R7 - R2 = R7 - 1 push n - 1
37: A370   M[R7+0] <- R3
38: 8630   R6 <- PC, PC <- 30     call sum
39: B201   R2 <- 1                pop n - 1
3A: 1772   R7 <- R7 + R2 = R7 + 1
3B: 9271   R2 <- M[R7+1]          R2 <- n
3C: 1112   R1 <- R1 + R2          R1 <- sum(n-1) + n
3D: 9670   R6 <- M[R7+0]          pop return address
3E: B201   R2 <- 1
3F: 1772   R7 <- R7 + R2 = R7 + 1
40: 7600   PC <- R6               return
```

Implementing Recursive Functions, cont'd

- **Main program makes the first call**

```
00:  B000     R0 <- 0                    R0 holds 0
01:  B7FF     R7 <- FF                   initialize stack pointer
02:  B210     R2 <- 50                       R2 <- address of n
03:  9220     R2 <- M[R2+0]                  R2 <- n
04:  B101     R1 <- 1                        push n
05:  2771     R7 <- R7 - R1 = R7 - 1
06:  A270     M[R7+0] <- R2
07:  8630     R6 <- PC, PC <- 30             call sum
08:  B201     R2 <- 1                        pop n
09:  1772     R7 <- R7 + R2 = R7 + 1
0A:  4102     print R1                       print sum(n)
0B:  0000     halt

50:  0000                                    n
```