Chapter 20: Advanced Sorting Algorithms

Data Structures and Algorithms with C

Quicksort | Average case | Non-recursive | Median-of-3 | qsort | Merge sort | Heap sort | Radix sort

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/dsa_c/slides/Chapter_20_Advanced_Sorting_Algorithms.html

Learning Outcomes

Quicksort

Explain the *partition* invariant; trace `partition()` and the recursion step by step.

Analysis

Derive best n lg n, worst n^2 and average ≈ 1.39 n lg n from recurrences.

Improvements

A non-recursive version (O(lg n) stack), median-of-3, insertion sort for small pieces; `qsort` in C.

Merge sort

Trace merge, the stack frames and the linked (index-based) version; O(n) extra memory.

Heap sort

Build a heap bottom-up in O(n) and sort in place in O(n lg n).

By digits

Implement LSD radix sort with linked-list buckets and interpret its cost.

Quicksort

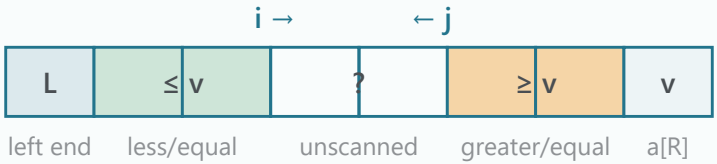
- Developed by **C. A. R. (Tony) Hoare** in 1962.
- **Divide and conquer**: split the problem recursively into subproblems until they are small enough to solve directly.
- Time proportional to n lg n on average; n^2 in the worst case. Uses only a small auxiliary **stack**.
- **Not stable**, yet one of the fastest general-purpose algorithms known. The standard C library has `qsort`.

```
int partition(int a[], int L, int R);

void quicksort1(int a[], int L, int R) {
    if (R <= L) return;
    int i = partition(a, L, R);
    quicksort1(a, L, i - 1);
    quicksort1(a, i + 1, R);
}
```

INSTRUCTOR "This code is taken from Sedgewick's book. Very nicely written code. (Sedgewick was Knuth's PhD student and worked a lot on quicksort.)"

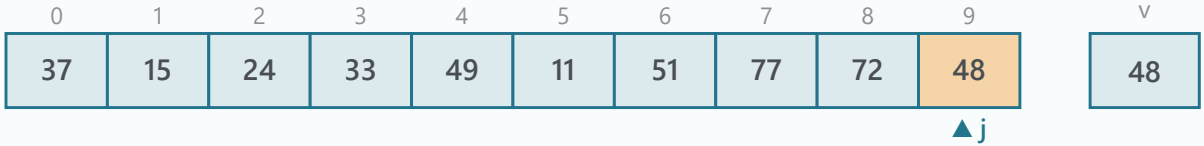
partition()




```
int partition(int a[], int L, int R) {
    int i = L - 1, j = R, v = a[R]; // pivot a[R]
    for (;;) {
        while (less(a[++i], v))
            ;
        while (less(v, a[--j]))
            if (j == L) break;
        if (i >= j) break;
        SWAP(a[i], a[j]);
    }
    SWAP(a[i], a[R]);
    return i;
}
```

- [2] The pivot (*partitioning element*) is `v = a[R]`.
- [4] i stops at the first element from the left that is **greater than or equal to** v. No bounds test is needed: at worst `a[R] = v` stops it at i = R.
- [6–7] j stops at an element from the right that is **less than or equal to** v; `j == L` keeps this loop in bounds.
- [9] Both elements are on the wrong side → swap. [8] When i and j meet, the loop ends; [11] the pivot goes to a[i]: left ≤ v ≤ right.

Code Trace: `partition(a, 0, 9)`



Pivot **v = a[9] = 48**. i = L − 1 = −1, j = R = 9: the scans start with ++i and --j.

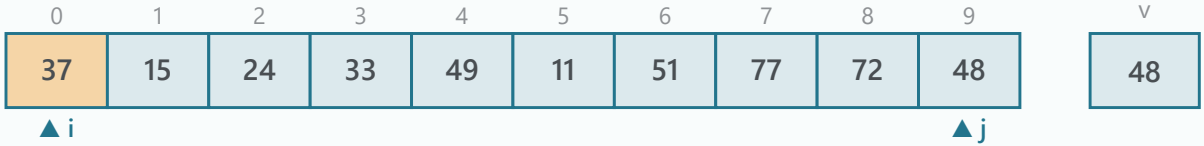
Step 1 / 15

```
1  int partition(int a[], int L, int R) {
▶      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`L = 0` `R = 9` `i = -1` `j = 9` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[0] = 37 < 48 → i keeps going.

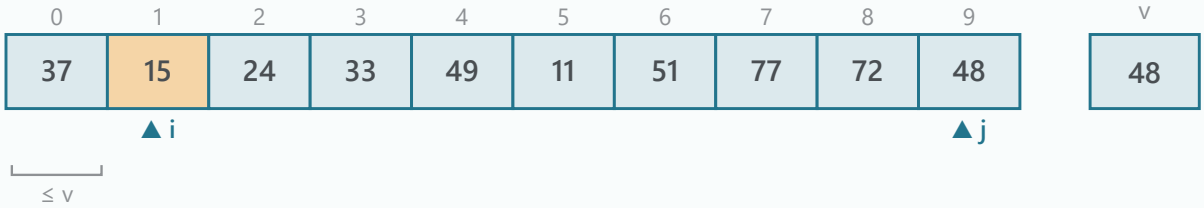
Step 2 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 0` `j = 9` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[1] = 15 < 48 → i keeps going.

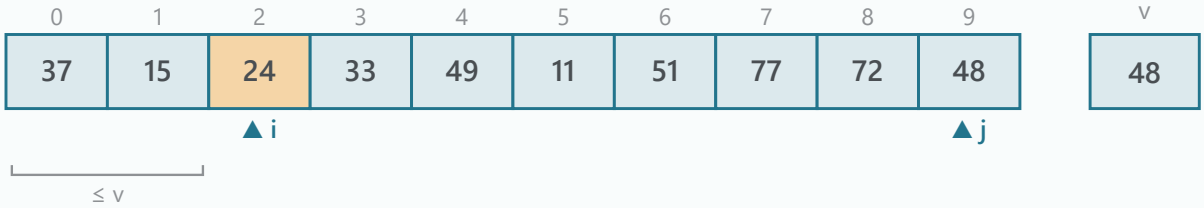
Step 3 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 1` `j = 9` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[2] = 24 < 48 → i keeps going.

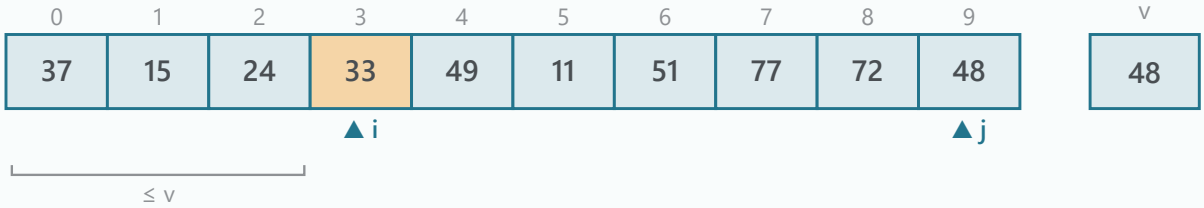
Step 4 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 2` `j = 9` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[3] = 33 < 48 → i keeps going.

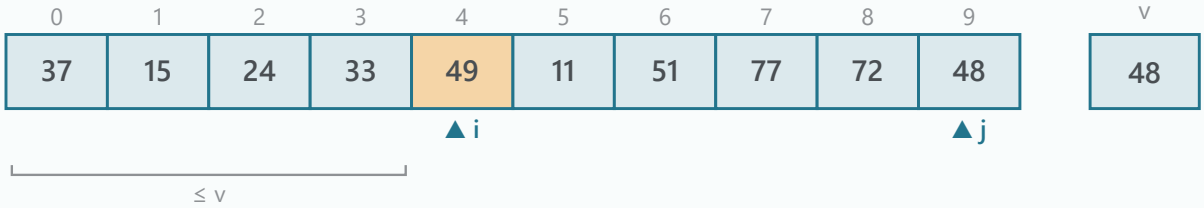
Step 5 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 3` `j = 9` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[4] = 49 ≥ 48 → stops at **i = 4** (an element that belongs on the right).

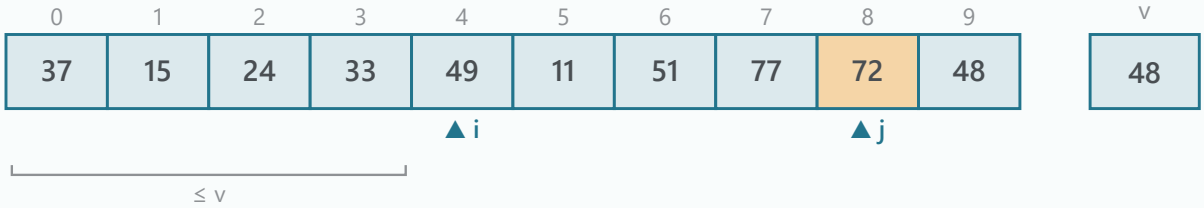
Step 6 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 4` `j = 9` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[8] = 72 > 48 → j moves left.

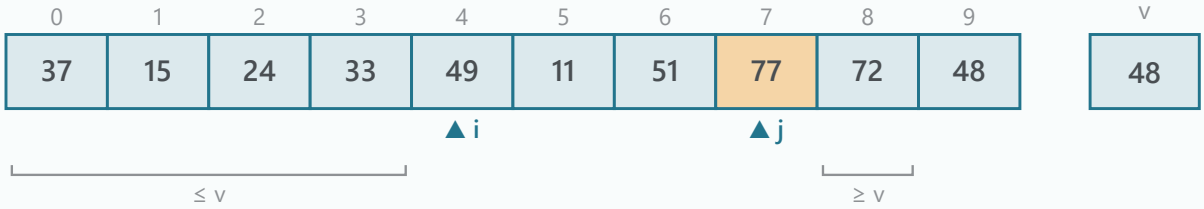
Step 7 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (less(a[++i], v))
5              ;
▶          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 4` `j = 8` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[7] = 77 > 48 → j moves left.

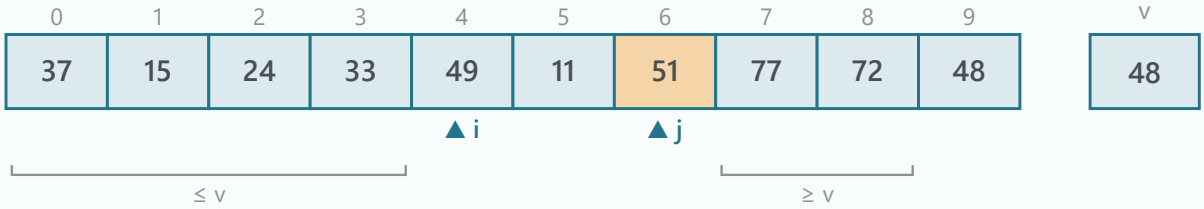
Step 8 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (less(a[++i], v))
5              ;
▶          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 4` `j = 7` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[6] = 51 > 48 → j moves left.

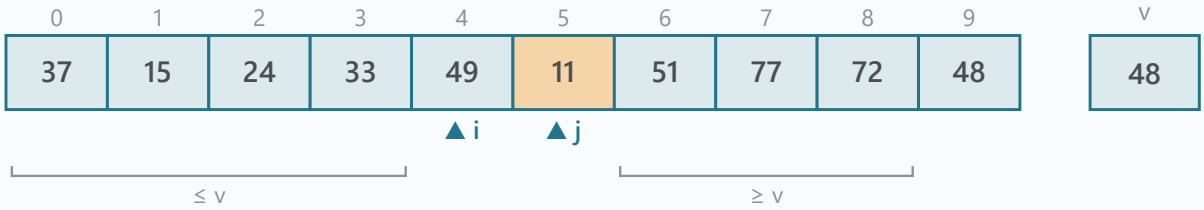
Step 9 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (less(a[++i], v))
5              ;
▶          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 4` `j = 6` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[5] = 11 ≤ 48 → stops at **j = 5**.

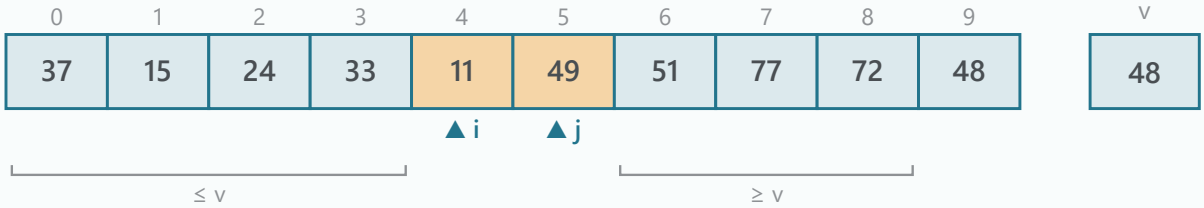
Step 10 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (less(a[++i], v))
5              ;
▶          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 4` `j = 5` `v = 48`

Code Trace: `partition(a, 0, 9)`



i < j: a[4] and a[5] are on the wrong sides → swap: [37, 15, 24, 33, 11, 49, 51, 77, 72, 48].

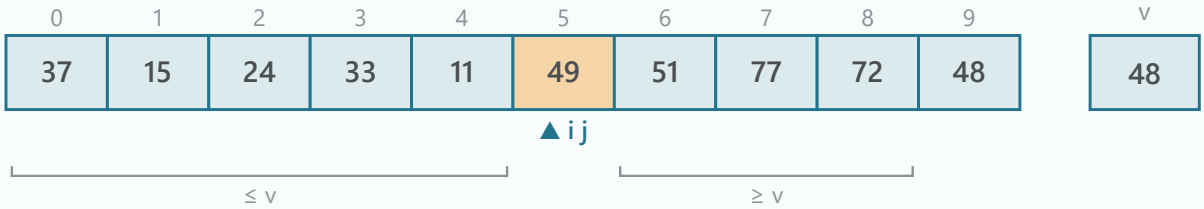
Step 11 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
▶          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 4` `j = 5` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[5] = 49 ≥ 48 → stops at **i = 5** (an element that belongs on the right).

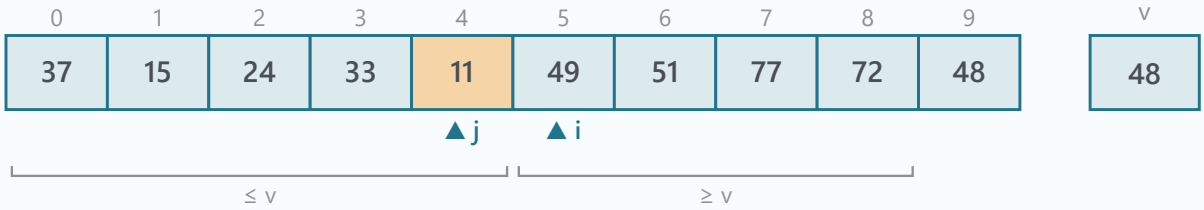
Step 12 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
▶          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 5` `j = 5` `v = 48`

Code Trace: `partition(a, 0, 9)`



a[4] = 11 ≤ 48 → stops at **j = 4**.

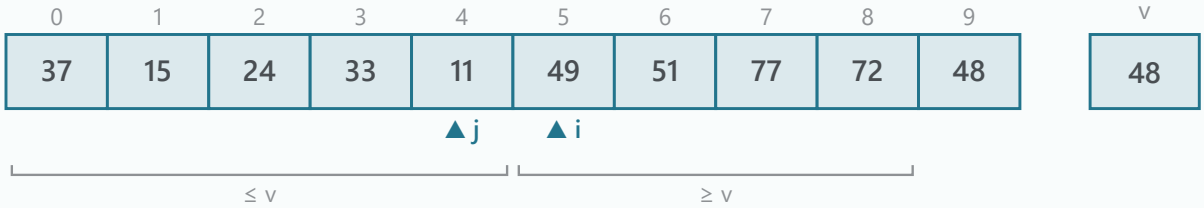
Step 13 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 5` `j = 4` `v = 48`

Code Trace: `partition(a, 0, 9)`



i = 5 ≥ j = 4: the indices crossed → leave the loop.

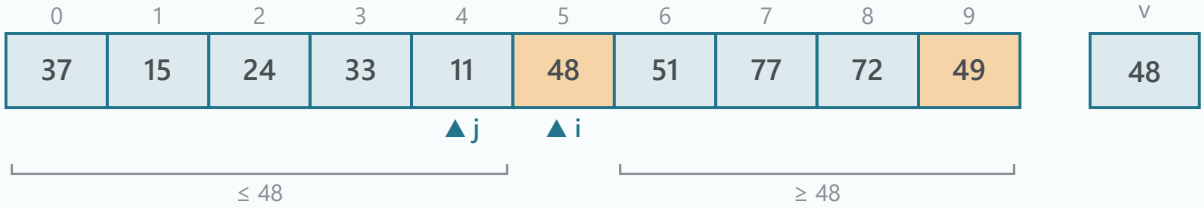
Step 14 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
▶          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
11     SWAP(a[i], a[R]);
12     return i;
13 }
```

VARIABLES

`i = 5` `j = 4` `v = 48`

Code Trace: `partition(a, 0, 9)`



The pivot moves to a[5] and **5 is returned**. In the sorted array, 48 would sit exactly here.

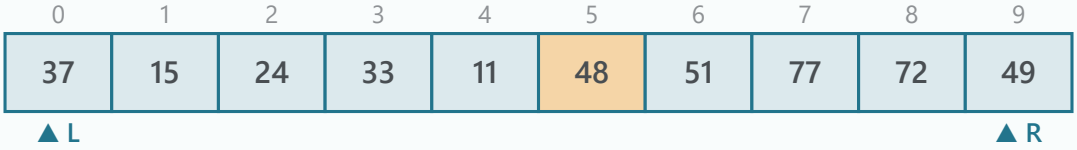
Step 15 / 15

```
1  int partition(int a[], int L, int R) {
2      int i = L - 1, j = R, v = a[R]; // pivot a[R]
3      for (;;) {
4          while (less(a[++i], v))
5              ;
6          while (less(v, a[--j]))
7              if (j == L) break;
8          if (i >= j) break;
9          SWAP(a[i], a[j]);
10     }
▶      SWAP(a[i], a[R]);
▶      return i;
13 }
```

VARIABLES

`i = 5` `j = 4` `v = 48`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



partition(0, 9): pivot 48 → position **5**. Left a[0..4] and right a[6..9] are sorted separately.

Step 1 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

VARIABLES

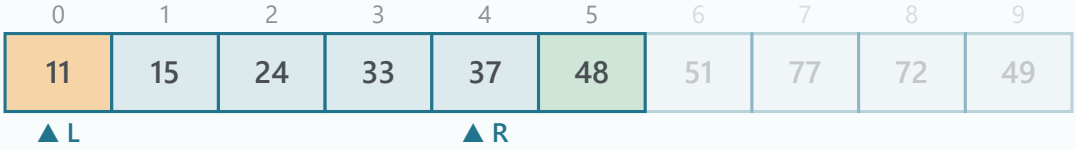
`L = 0` `R = 9` `i = 5`

CALL STACK

`quicksort1(0, 9)`

`main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



partition(0, 4): pivot 11 → position **0**. Left a[0..-1] and right a[1..4] are sorted separately.

Step 2 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

VARIABLES

`L = 0` `R = 4` `i = 0`

CALL STACK

- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



quicksort1(0, -1): empty range → return.

Step 3 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

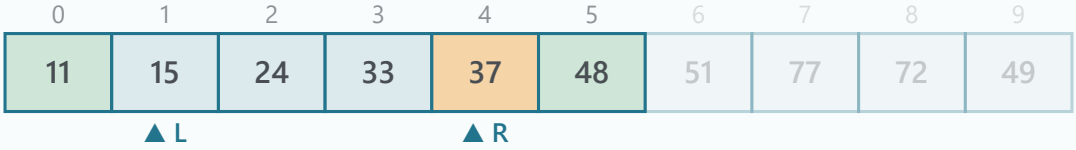
VARIABLES

L = 0 R = -1

CALL STACK

- quicksort1(0, -1)
- quicksort1(0, 4)
- quicksort1(0, 9)
- main()

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



partition(1, 4): pivot 37 → position **4**. Left a[1..3] and right a[5..4] are sorted separately.

Step 4 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

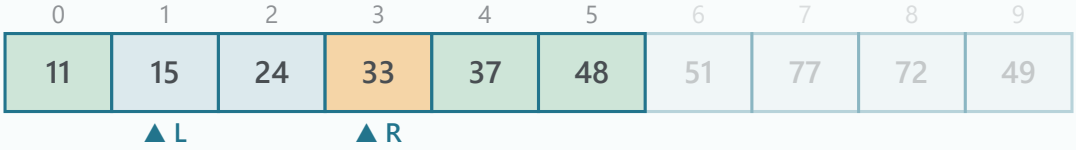
VARIABLES

`L = 1` `R = 4` `i = 4`

CALL STACK

- `quicksort1(1, 4)`
- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



partition(1, 3): pivot 33 → position **3**. Left a[1..2] and right a[4..3] are sorted separately.

Step 5 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

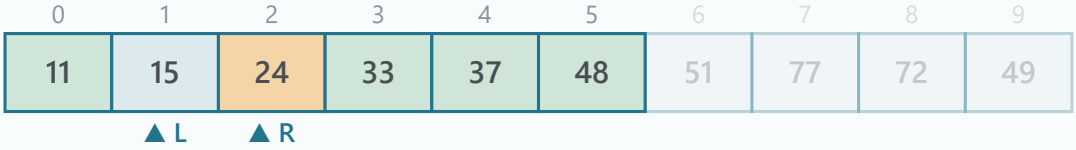
VARIABLES

`L = 1` `R = 3` `i = 3`

CALL STACK

- `quicksort1(1, 3)`
- `quicksort1(1, 4)`
- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



partition(1, 2): pivot 24 → position **2**. Left a[1..1] and right a[3..2] are sorted separately.

Step 6 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

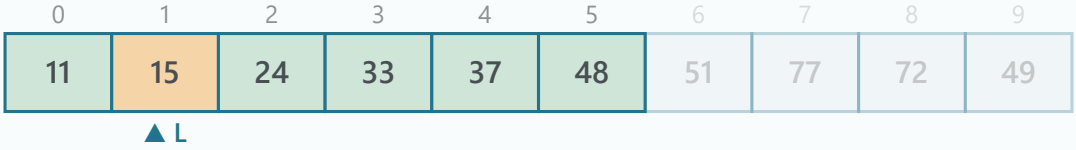
VARIABLES

`L = 1` `R = 2` `i = 2`

CALL STACK

- `quicksort1(1, 2)`
- `quicksort1(1, 3)`
- `quicksort1(1, 4)`
- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



quicksort1(1, 1): one element, already in place → return.

Step 7 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

VARIABLES

L = 1 R = 1

CALL STACK

- `quicksort1(1, 1)`
- `quicksort1(1, 2)`
- `quicksort1(1, 3)`
- `quicksort1(1, 4)`
- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack

0	1	2	3	4	5	6	7	8	9
11	15	24	33	37	48	51	77	72	49

quicksort1(3, 2): empty range → return.

Step 8 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

VARIABLES

L = 3 R = 2

CALL STACK

- **quicksort1(3, 2)**
- quicksort1(1, 2)
- quicksort1(1, 3)
- quicksort1(1, 4)
- quicksort1(0, 4)
- quicksort1(0, 9)
- main()

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



quicksort1(4, 3): empty range → return.

Step 9 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

VARIABLES

L = 4 R = 3

CALL STACK

- `quicksort1(4, 3)`
- `quicksort1(1, 3)`
- `quicksort1(1, 4)`
- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack

0	1	2	3	4	5	6	7	8	9
11	15	24	33	37	48	51	77	72	49

quicksort1(5, 4): empty range → return.

Step 10 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

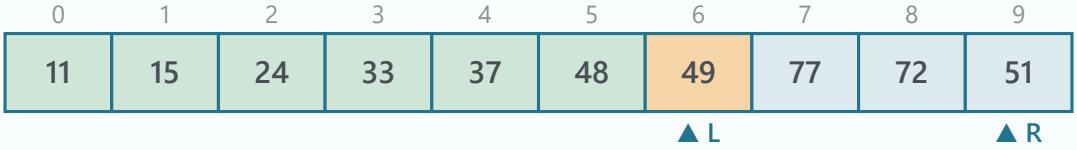
VARIABLES

L = 5 R = 4

CALL STACK

- `quicksort1(5, 4)`
- `quicksort1(1, 4)`
- `quicksort1(0, 4)`
- `quicksort1(0, 9)`
- `main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



partition(6, 9): pivot 49 → position **6**. Left a[6..5] and right a[7..9] are sorted separately.

Step 11 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

VARIABLES

`L = 6` `R = 9` `i = 6`

CALL STACK

- `quicksort1(6, 9)`
- `quicksort1(0, 9)`
- `main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



quicksort1(6, 5): empty range → return.

Step 12 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

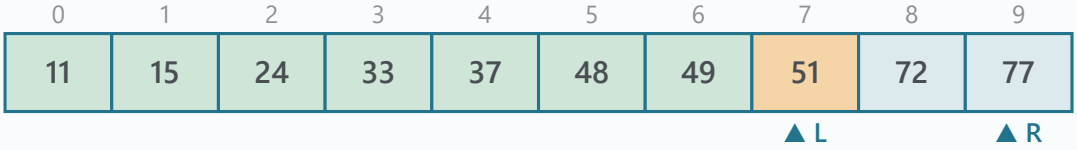
VARIABLES

L = **6** R = **5**

CALL STACK

- **quicksort1(6, 5)**
- quicksort1(6, 9)
- quicksort1(0, 9)
- main()

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



partition(7, 9): pivot 51 → position **7**. Left a[7..6] and right a[8..9] are sorted separately.

Step 13 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

VARIABLES

`L = 7` `R = 9` `i = 7`

CALL STACK

- `quicksort1(7, 9)`
- `quicksort1(6, 9)`
- `quicksort1(0, 9)`
- `main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack

0	1	2	3	4	5	6	7	8	9
11	15	24	33	37	48	49	51	72	77

quicksort1(7, 6): empty range → return.

Step 14 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

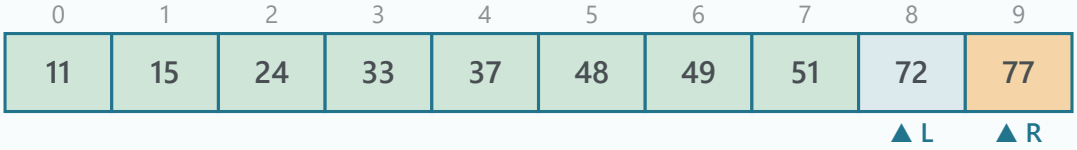
VARIABLES

L = 7 R = 6

CALL STACK

- **quicksort1(7, 6)**
- quicksort1(7, 9)
- quicksort1(6, 9)
- quicksort1(0, 9)
- main()

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



partition(8, 9): pivot 77 → position **9**. Left a[8..8] and right a[10..9] are sorted separately.

Step 15 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
4     if (R <= L) return;
▶     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

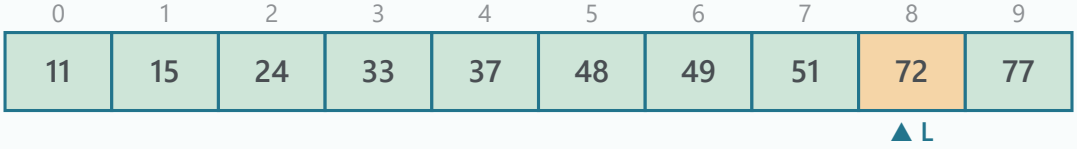
VARIABLES

`L = 8` `R = 9` `i = 9`

CALL STACK

- `quicksort1(8, 9)`
- `quicksort1(7, 9)`
- `quicksort1(6, 9)`
- `quicksort1(0, 9)`
- `main()`

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



quicksort1(8, 8): one element, already in place → return.

Step 16 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

VARIABLES

L = 8 R = 8

CALL STACK

- quicksort1(8, 8)
- quicksort1(8, 9)
- quicksort1(7, 9)
- quicksort1(6, 9)
- quicksort1(0, 9)
- main()

Code Trace: `quicksort1(a, 0, 9)`, the Call Stack



quicksort1(10, 9): empty range → return.

Step 17 / 17

```
1 int partition(int a[], int L, int R);
2
3 void quicksort1(int a[], int L, int R) {
▶     if (R <= L) return;
5     int i = partition(a, L, R);
6     quicksort1(a, L, i - 1);
7     quicksort1(a, i + 1, R);
8 }
```

VARIABLES

L = 10 R = 9

CALL STACK

- quicksort1(10, 9)
- quicksort1(8, 9)
- quicksort1(7, 9)
- quicksort1(6, 9)
- quicksort1(0, 9)
- main()

Assessment: Why Is It Quick?

Right position + smaller problem

Bubble and selection sort also put one element in place at each step, but the next subproblem still has n − 1 elements. Quicksort places the pivot and also splits the problem into **two smaller pieces**.

Recursion tree

The `quicksort1(a, 0, 9)` trace makes 17 calls for 10 elements: every non-trivial call put a pivot in its final place (green cells in the trace).

INSTRUCTOR "You may say that bubble or selection sort also find the right position of one element at each step. Yes, they do, but they cannot help the next step by shrinking the problem."

INSTRUCTOR "I will write three or four quicksort algorithms, so understanding this one matters. [...] Small pieces of code should be written over and over, in different ways."

Reading: Algorithms and Computers

INSTRUCTOR "— Sir! Where did this man try quicksort in the 1960s, how did he test it? Where does the word algorithm come from? Did al-Khwarizmi have a 4 GB laptop in the 9th century? ☺"

INSTRUCTOR "An algorithm means that someone who knows nothing about it can take it, run it step by step and get the right result. Besides, there were computers in the 1960s. But our course is independent of computers. Publishing a paper on an algorithm is enough."

Al-Khwarizmi (780–850)

Mathematician and astronomer at the House of Wisdom in Baghdad. Author of *al-Kitāb al-Mukhtaṣar fī Ḥisāb al-Jabr wa'l-Muqābala*, which gave systematic solutions of linear and quadratic equations.

Origin of the words

"Algebra" comes from the book's *al-jabr* method, and "algorithm" from *Algoritmi*, the Latin form of his name. His works introduced the decimal positional number system to the West.

Quicksort: Worst and Best Case

Worst: a sorted array

The pivot a[R] is always the largest and stays put; the next call is one element smaller:
$n + (n-1) + \ldots + 1 = n(n+1)/2 = O(n^2)$

Best: splitting exactly in half

At each step i and j advance n steps in total, leaving two subproblems of size n/2:
$C_n = 2C_{n/2} + n = 2^k C_{n/2^k} + k \cdot n = n \lg n$

INSTRUCTOR “Now you'll be wondering: ‘What's so quick about it?’”

The best case is n lg n; the optimal sorting bound was n lg n − 1.44n. So the leading coefficient is still 1, with an excess of 1.44n.

Counts: quicksort1 on Sorted and Random Input

n	sorted: comparisons	depth	random: comparisons	depth	n lg n
10	63	9	36	5	33
100	5,148	99	721	9	664
1000	501,498	999	12,831	25	9,966

Comparisons inside partition() counted by a Python simulation (random input: seed 20).

Sorted input

≈ $n^2/2$ comparisons and recursion depth n − 1: in C, a risk of **stack overflow** for large n.

Random input

Comparisons of order n lg n, depth only a few lg n.

Average-Case Analysis: The Recurrence

$$C_n = (n+1) + (1/n)\,\Sigma_{k=1}^{n}\,(C_{k-1} + C_{n-k}),\quad C_0 = C_1 = 0$$

- The split point k, $1 \le k \le n$, is **equally likely**: running partition() on all n! orderings gives equal frequencies.
- n + 1: the comparisons with the pivot in partition(); a split at k leaves subarrays of (k−1) and (n−k) elements.
- The two sums contain the same terms: $\Sigma(C_{k-1} + C_{n-k}) = 2\,\Sigma_{k=0}^{n-1}\,C_k \Rightarrow n \cdot C_n = n(n+1) + 2\,\Sigma\,C_{k-1}$ … (1)

INSTRUCTOR "Toss a coin and count the heads. 10 times, 100 times, 1000 times, even a million. Someone said: it converges to ½! That is how average-case analysis tells us where things concentrate."

INSTRUCTOR "'In my opinion, every k is equally likely.' There's no 'my opinion, your opinion' here! There's mathematics."

Average Case: The Subtraction Trick and the Result

- Replace n by n − 1 in (1): $(n-1)C_{n-1} = n(n-1) + 2\,\Sigma_{k=1}^{n-1} C_{k-1}$ … (2)
- (1) − (2): $n\cdot C_n - (n-1)C_{n-1} = 2n + 2C_{n-1} \Rightarrow n\cdot C_n = (n+1)C_{n-1} + 2n$
- (dividing by $n(n+1)$) $C_n/(n+1) = C_{n-1}/n + 2/(n+1)$: now it unrolls by iteration.
- Unrolling gives $C_n \approx 2(n+1)(1/4 + \ldots + 1/(n+1)) \le 2(n+1)\ln n$.

$$C_n \approx 2n \ln n = 2 \ln 2 \cdot n \lg n \approx 1.39\ n \lg n$$

n	C_n (from the recurrence)	1.39 n lg n	ratio
10	37.1	46.2	0.80
100	780.5	923.5	0.85
1000	12,318.6	13,852.4	0.89

Table: the recurrence computed exactly in Python. The ratio slowly approaches 1 (a lower-order −cn term).

INSTRUCTOR “The book solves this by substitution. We will use a smarter trick.”

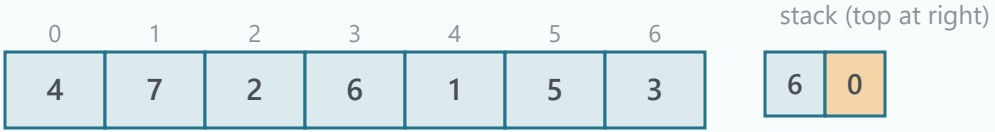
Non-Recursive Quicksort (quicksort2)

```
void quicksort2(int a[], int L, int R) {
  stack_init(); push2(L, R);
  while (!stack_empty()) {
    L = pop(); R = pop();
    if (R <= L) continue;
    int i = partition(a, L, R);
    if (i - L > R - i) {       // larger part first
      push2(L, i - 1); push2(i + 1, R);
    } else {
      push2(i + 1, R); push2(L, i - 1);
    }
  }
}
```

```
#define STACK_MAX 64  /* > 2·lg n */
static int stack[STACK_MAX], top;
void stack_init(void) { top = 0; }
int stack_empty(void) { return top == 0; }
void push(int x) { stack[top++] = x; }
int pop(void) { return stack[--top]; }
#define push2(A, B) \
  do { push(B); push(A); } while (0)
```

- Goal: use our own stack instead of the system stack. `push2(A,B)` pushes B first, then A, so L is popped first.
- [7–11] The **larger part is pushed first**, the smaller one last: the last pushed is popped first, so the smaller part is always processed next → the stack depth stays **O(lg n)** (at most ≈ lg n pairs).
- The memory recurrence $T_n = T_{n/2} + 2 \Rightarrow T_n = 2 \lg n$ (as in binary search).

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack



The first problem (0, 6) is pushed: first R = 6, then L = 0.

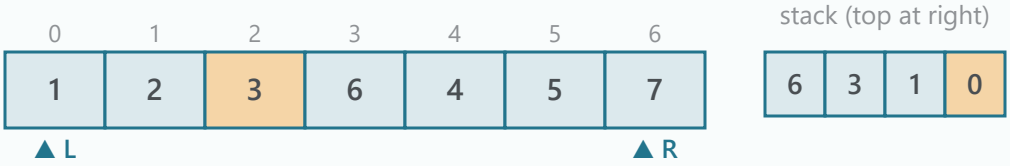
Step 1 / 11

```
1  void quicksort2(int a[], int L, int R) {
▶    stack_init(); push2(L, R);
3    while (!stack_empty()) {
4      L = pop(); R = pop();
5      if (R <= L) continue;
6      int i = partition(a, L, R);
7      if (i - L > R - i) {       // larger part first
8        push2(L, i - 1); push2(i + 1, R);
9      } else {
10       push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`stack = [6, 0]`

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack



(0, 6): partition → pivot 3 at position 2. The right part (3,6) is not smaller → pushed first, then the left (0,1); the smaller part is on top.

Step 2 / 11

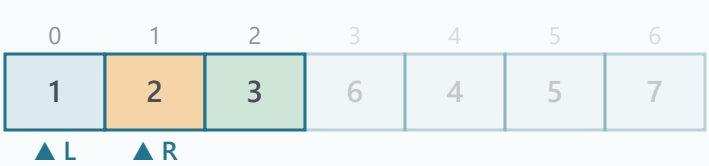
```
1  void quicksort2(int a[], int L, int R) {
2    stack_init(); push2(L, R);
3    while (!stack_empty()) {
4      L = pop(); R = pop();
5      if (R <= L) continue;
6      int i = partition(a, L, R);
7      if (i - L > R - i) {       // larger part first
8        push2(L, i - 1); push2(i + 1, R);
▶      } else {
▶        push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`L = 0` `R = 6` `i = 2`

`stack = [6, 3, 1, 0]`

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack





(0, 1): partition → pivot 2 at position 1. The left part (0,0) is larger → pushed first, then the right (2,1); the smaller part is on top.

Step 3 / 11

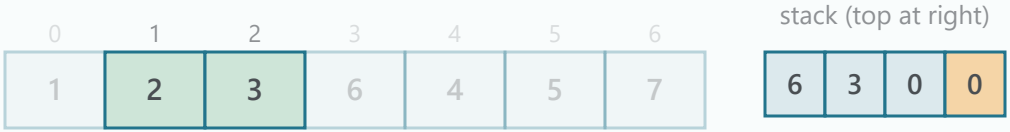
```
1  void quicksort2(int a[], int L, int R) {
2    stack_init(); push2(L, R);
3    while (!stack_empty()) {
4      L = pop(); R = pop();
5      if (R <= L) continue;
6      int i = partition(a, L, R);
▶      if (i - L > R - i) {       // larger part first
▶        push2(L, i - 1); push2(i + 1, R);
9      } else {
10       push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`L = 0` `R = 1` `i = 1`

`stack = [6, 3, 0, 0, 1, 2]`

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack



(2, 1) popped: empty range → continue.

Step 4 / 11

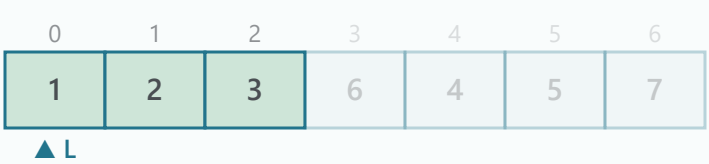
```
1  void quicksort2(int a[], int L, int R) {
2    stack_init(); push2(L, R);
3    while (!stack_empty()) {
▶      L = pop(); R = pop();
▶      if (R <= L) continue;
6      int i = partition(a, L, R);
7      if (i - L > R - i) {       // larger part first
8        push2(L, i - 1); push2(i + 1, R);
9      } else {
10       push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`L = 2` `R = 1`

`stack = [6, 3, 0, 0]`

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack



(0, 0) popped: one element → continue.

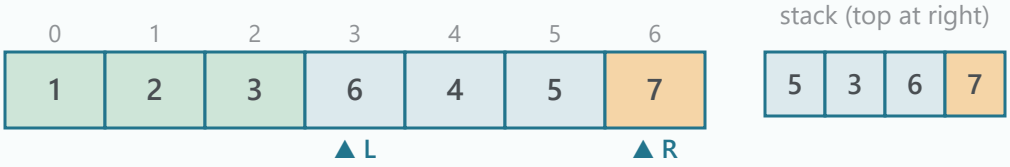
Step 5 / 11

```
1  void quicksort2(int a[], int L, int R) {
2    stack_init(); push2(L, R);
3    while (!stack_empty()) {
▶      L = pop(); R = pop();
▶      if (R <= L) continue;
6      int i = partition(a, L, R);
7      if (i - L > R - i) {       // larger part first
8        push2(L, i - 1); push2(i + 1, R);
9      } else {
10       push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`L = 0` `R = 0` `stack = [6, 3]`

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack



(3, 6): partition → pivot 7 at position 6. The left part (3,5) is larger → pushed first, then the right (7,6); the smaller part is on top.

Step 6 / 11

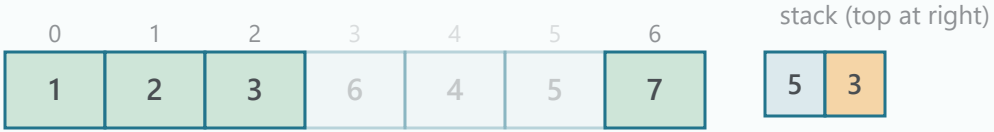
```
1  void quicksort2(int a[], int L, int R) {
2    stack_init(); push2(L, R);
3    while (!stack_empty()) {
4      L = pop(); R = pop();
5      if (R <= L) continue;
6      int i = partition(a, L, R);
▶      if (i - L > R - i) {       // larger part first
▶        push2(L, i - 1); push2(i + 1, R);
9      } else {
10       push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`L = 3` `R = 6` `i = 6`

`stack = [5, 3, 6, 7]`

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack



(7, 6) popped: empty range → continue.

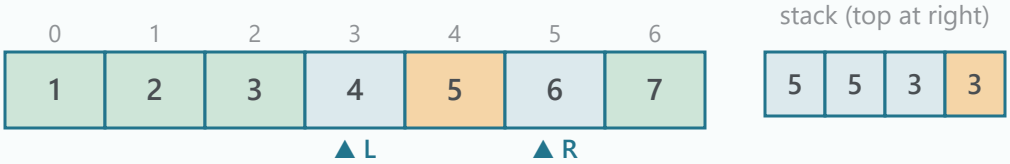
Step 7 / 11

```
1  void quicksort2(int a[], int L, int R) {
2    stack_init(); push2(L, R);
3    while (!stack_empty()) {
▶      L = pop(); R = pop();
▶      if (R <= L) continue;
6      int i = partition(a, L, R);
7      if (i - L > R - i) {       // larger part first
8        push2(L, i - 1); push2(i + 1, R);
9      } else {
10       push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`L = 7` `R = 6` `stack = [5, 3]`

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack



(3, 5): partition → pivot 5 at position 4. The right part (5,5) is not smaller → pushed first, then the left (3,3); the smaller part is on top.

Step 8 / 11

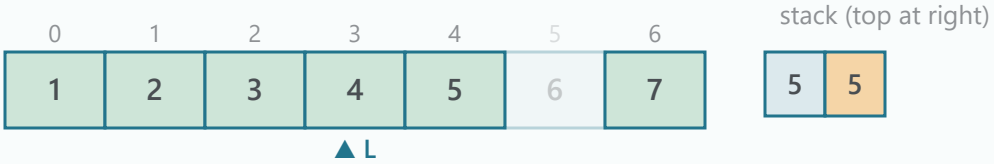
```
 1 void quicksort2(int a[], int L, int R) {
 2   stack_init(); push2(L, R);
 3   while (!stack_empty()) {
 4     L = pop(); R = pop();
 5     if (R <= L) continue;
 6     int i = partition(a, L, R);
 7     if (i - L > R - i) {       // larger part first
 8       push2(L, i - 1); push2(i + 1, R);
▶      } else {
▶        push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`L = 3` `R = 5` `i = 4`

`stack = [5, 5, 3, 3]`

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack



(3, 3) popped: one element → continue.

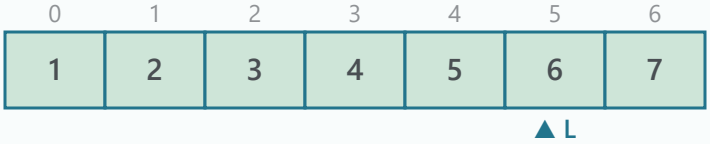
Step 9 / 11

```
1  void quicksort2(int a[], int L, int R) {
2    stack_init(); push2(L, R);
3    while (!stack_empty()) {
▶      L = pop(); R = pop();
▶      if (R <= L) continue;
6      int i = partition(a, L, R);
7      if (i - L > R - i) {       // larger part first
8        push2(L, i - 1); push2(i + 1, R);
9      } else {
10       push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`L = 3` `R = 3` `stack = [5, 5]`

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack



stack (top at right)

empty

(5, 5) popped: one element → continue.

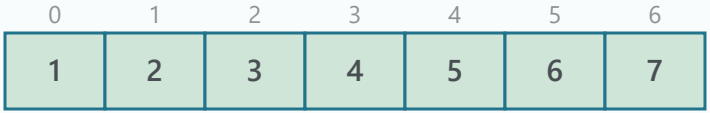
Step 10 / 11

```
1  void quicksort2(int a[], int L, int R) {
2    stack_init(); push2(L, R);
3    while (!stack_empty()) {
▶      L = pop(); R = pop();
▶      if (R <= L) continue;
6      int i = partition(a, L, R);
7      if (i - L > R - i) {       // larger part first
8        push2(L, i - 1); push2(i + 1, R);
9      } else {
10       push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`L = 5` `R = 5` `stack = []`

Code Trace: `quicksort2(a, 0, 6)`, an Explicit Stack



stack (top at right)

empty

The stack is empty → done: [1, 2, 3, 4, 5, 6, 7]. At most 3 (L, R) pairs waited on the stack.

Step 11 / 11

```
1  void quicksort2(int a[], int L, int R) {
2    stack_init(); push2(L, R);
▶    while (!stack_empty()) {
4      L = pop(); R = pop();
5      if (R <= L) continue;
6      int i = partition(a, L, R);
7      if (i - L > R - i) {       // larger part first
8        push2(L, i - 1); push2(i + 1, R);
9      } else {
10       push2(i + 1, R); push2(L, i - 1);
11     }
12   }
13 }
```

VARIABLES

`stack = []`

Improved Quicksort: Median-of-3 + Insertion Sort

```c
#define M 10    /* leave pieces smaller than this */
void quicksort3(int a[], int L, int R) {
    if (R - L <= M) return;
    SWAP(a[(L + R) / 2], a[R - 1]);
    if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-1]
    if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
    if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <= a[R]
    int i = partition(a, L + 1, R - 1);
    quicksort3(a, L, i - 1);
    quicksort3(a, i + 1, R);
}
void sort(int a[], int L, int R) {
    quicksort3(a, L, R);
    insertion(a, L, R);   /* the small pieces left */
}
```

`if (less(A, B)) SWAP(A, B)` means "swap if A < B"; with the arguments in the order `(a[L], a[R-1])` it puts the larger one in a[L] and breaks the partition (tested with random arrays: an element larger than the pivot stays on its left). So the arguments are given in reverse order: the result is **a[L] ≤ a[R−1] ≤ a[R]**.

INSTRUCTOR "The standard C library function qsort() is probably this."

Why Median-of-3 and a Small-Piece Threshold?

Median-of-three

a[L], the middle element and a[R] are put in order; their median moves to a[R−1] and becomes the pivot. Quicksort did worst and insertion sort best on sorted data: the n^2 case on sorted input disappears.

Natural sentinels

Since a[L] ≤ pivot ≤ a[R], partitioning runs on L+1..R−1; the end elements stop the scans.

Threshold M

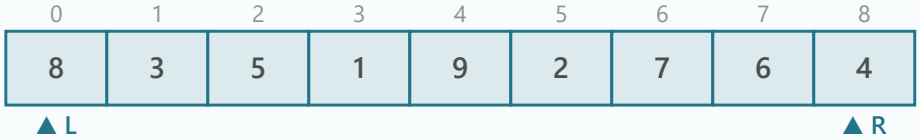
Pieces with R − L ≤ M are left unsorted; one final `insertion` pass in `sort()` fixes them all in nearly linear time. About 10% faster for 5 ≤ M ≤ 25.

Its limit

Median-of-3 rescues sorted and reverse-sorted input, but specially crafted bad inputs can still cause n^2 → the cure is introsort.

INSTRUCTOR “It never leaves you facing n^2. The analysis is a bit heavy…”

Code Trace: Median-of-3 Setup, `quicksort3(a, 0, 8)`



R − L = 8 > M = 3 (M = 3 in this trace) → the piece is large; partition it.

Step 1 / 7

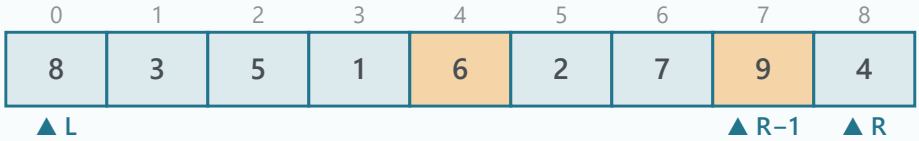
```
1  void quicksort3(int a[], int L, int R) {
▶      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-
   1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <=
   a[R]
7      int i = partition(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

VARIABLES

`L = 0` `R = 8` `M = 3`

`a = [8, 3, 5, 1, 9, 2, 7, 6, 4]`

Code Trace: Median-of-3 Setup, `quicksort3(a, 0, 8)`



The middle element a[4] is swapped with a[R−1] = a[7]: the candidate pivot 9 is now in a[7].

Step 2 / 7

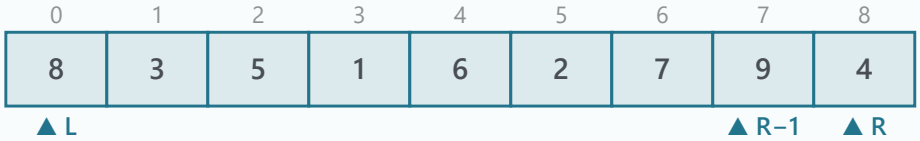
```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
▶      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-
   1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <=
   a[R]
7      int i = partition(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

VARIABLES

`L = 0` `R = 8` `M = 3`

`a = [8, 3, 5, 1, 6, 2, 7, 9, 4]`

Code Trace: Median-of-3 Setup, `quicksort3(a, 0, 8)`



a[7] = 9 ≥ a[0] = 8 → no swap.

Step 3 / 7

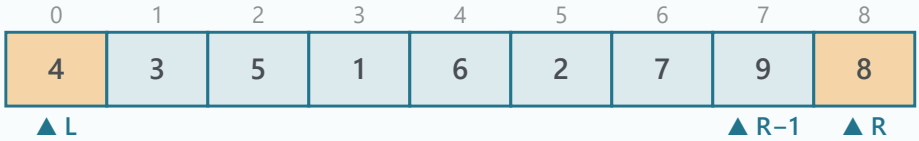
```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
▶      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <= a[R]
7      int i = partition(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

VARIABLES

`L = 0` `R = 8` `M = 3`

`a = [8, 3, 5, 1, 6, 2, 7, 9, 4]`

Code Trace: Median-of-3 Setup, `quicksort3(a, 0, 8)`



a[8] = 4 < a[0] = 8 → swap.

Step 4 / 7

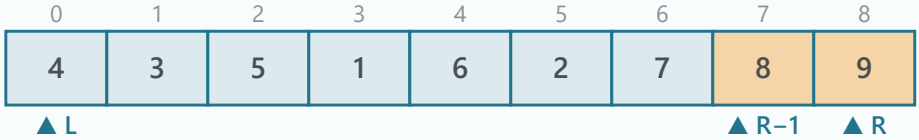
```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-1]
▶      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <= a[R]
7      int i = partition(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

VARIABLES

L = 0 | R = 8 | M = 3

a = [4, 3, 5, 1, 6, 2, 7, 9, 8]

Code Trace: Median-of-3 Setup, `quicksort3(a, 0, 8)`



a[8] = 8 < a[7] = 9 → swap.

Step 5 / 7

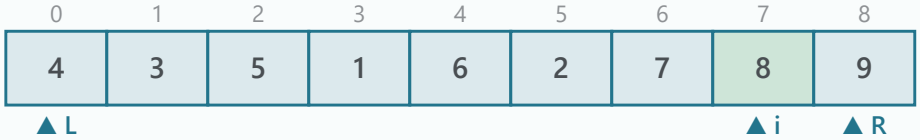
```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
▶      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <= a[R]
7      int i = partition(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

VARIABLES

L = 0 | R = 8 | M = 3

a = [4, 3, 5, 1, 6, 2, 7, 8, 9]

Code Trace: Median-of-3 Setup, `quicksort3(a, 0, 8)`



a[L] = 4 ≤ pivot 8 ≤ a[R] = 9. partition(a, 1, 7) → pivot at position **7**; left ≤ 8 ≤ right.

Step 6 / 7

```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-
   1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <=
   a[R]
▶      int i = partition(a, L + 1, R - 1);
8      quicksort3(a, L, i - 1);
9      quicksort3(a, i + 1, R);
10 }
```

VARIABLES

L = 0 | R = 8 | M = 3

a = [4, 3, 5, 1, 6, 2, 7, 8, 9]

i = 7

Code Trace: Median-of-3 Setup, `quicksort3(a, 0, 8)`



The left a[0..6] comes first: R − L = 6 > M, so median-of-3 repeats; partition(a, 1, 5) → 3 and the array becomes [1, 3, 2, 4, 6, 5, 7, 8, 9]. The remaining pieces (R − L = 2 ≤ M) return unsorted; finally the right a[8..8] (R − L = 0 ≤ M) returns. The insertion pass in sort() finishes the job.

Step 7 / 7

```
1  void quicksort3(int a[], int L, int R) {
2      if (R - L <= M) return;
3      SWAP(a[(L + R) / 2], a[R - 1]);
4      if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-
   1]
5      if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
6      if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <=
   a[R]
7      int i = partition(a, L + 1, R - 1);
▶      quicksort3(a, L, i - 1);
▶      quicksort3(a, i + 1, R);
10 }
```

VARIABLES

`L = 0` `R = 8` `M = 3`

`a = [4, 3, 5, 1, 6, 2, 7, 8, 9]`

`i = 7`

READING

Reading: Mandatory Internships

INSTRUCTOR “I believe internships were introduced out of necessity. In the old days you wrote a program, brought it to the lab and had it punched onto cards. You got one, at most two compile turns a day. How many times a day can you compile now? 50? 100?”

INSTRUCTOR “Now things are different… They use our students for errands: do this, do that. Like digging soil with a keyboard… We have nothing to do with the market! What matters to us is educating scientists…”

Code Trace: `qsort` and the Comparison Function



The array is ready; `sizeof a / sizeof a[0]` = 20 / 4 = 5 elements (assuming a 4-byte int).

Step 1 / 11

```
 1 int ascending(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
▶      int a[] = {37, 15, 24, 33, 49};
▶      size_t n = sizeof a / sizeof a[0];
 9     qsort(a, n, sizeof a[0], ascending);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

`a = [37, 15, 24, 33, 49]` `n = 5`

CALL STACK

`main()`

OUTPUT

Code Trace: `qsort` and the Comparison Function



`qsort` gets the array's address, the element count, the element size and the **address of the comparison function**. qsort does not know the element type; it works with `void *`.

Step 2 / 11

```
 1 int ascending(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 ▶     qsort(a, n, sizeof a[0], ascending);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

`a = [37, 15, 24, 33, 49]` `n = 5`

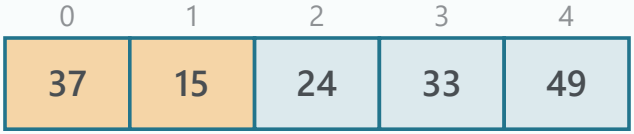
CALL STACK

`qsort(a, 5, 4, ascending)`

`main()`

OUTPUT

Code Trace: `qsort` and the Comparison Function



Sample call ascending(&37, &15) → **1**: positive = the first element goes after. (The standard does not say which pairs qsort compares, or in what order.)

Step 3 / 11

```
 1 int ascending(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 9     qsort(a, n, sizeof a[0], ascending);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

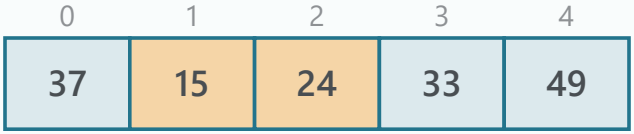
`x = 37` `y = 15` `ret = 1`

CALL STACK

- `ascending(p, q)`
- `qsort(a, 5, 4, ascending)`
- `main()`

OUTPUT

Code Trace: `qsort` and the Comparison Function



Sample call ascending(&15, &24) → **-1**: negative = the first element goes before. (The standard does not say which pairs qsort compares, or in what order.)

Step 4 / 11

```
1  int ascending(const void *p, const void *q) {
▶      int x = *(const int *)p;
▶      int y = *(const int *)q;
▶      return (x > y) - (x < y);  // -1, 0, +1
5  }
6  int main(void) {
7      int a[] = {37, 15, 24, 33, 49};
8      size_t n = sizeof a / sizeof a[0];
9      qsort(a, n, sizeof a[0], ascending);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

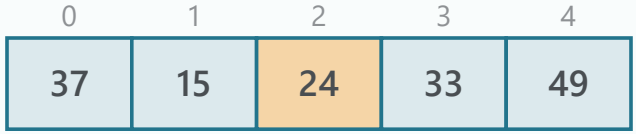
`x = 15` `y = 24` `ret = -1`

CALL STACK

- `ascending(p, q)`
- `qsort(a, 5, 4, ascending)`
- `main()`

OUTPUT

Code Trace: `qsort` and the Comparison Function



Sample call ascending(&24, &24) → **0**: zero = equal (relative order not guaranteed). (The standard does not say which pairs qsort compares, or in what order.)

Step 5 / 11

```
1  int ascending(const void *p, const void *q) {
▶      int x = *(const int *)p;
▶      int y = *(const int *)q;
▶      return (x > y) - (x < y);  // -1, 0, +1
5  }
6  int main(void) {
7      int a[] = {37, 15, 24, 33, 49};
8      size_t n = sizeof a / sizeof a[0];
9      qsort(a, n, sizeof a[0], ascending);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

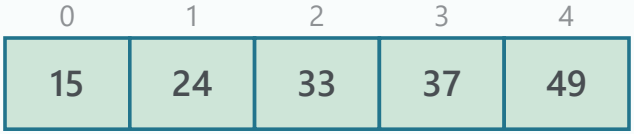
`x = 24` `y = 24` `ret = 0`

CALL STACK

- `ascending(p, q)`
- `qsort(a, 5, 4, ascending)`
- `main()`

OUTPUT

Code Trace: `qsort` and the Comparison Function



qsort returned: the array is in ascending order.

Step 6 / 11

```
1  int ascending(const void *p, const void *q) {
2      int x = *(const int *)p;
3      int y = *(const int *)q;
4      return (x > y) - (x < y);  // -1, 0, +1
5  }
6  int main(void) {
7      int a[] = {37, 15, 24, 33, 49};
8      size_t n = sizeof a / sizeof a[0];
▶      qsort(a, n, sizeof a[0], ascending);
10     for (size_t i = 0; i < n; i++)
11         printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

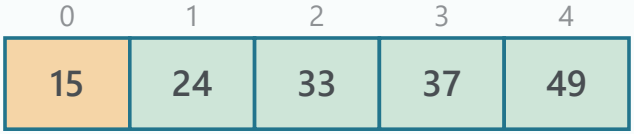
a = [15, 24, 33, 37, 49] n = 5

CALL STACK

main()

OUTPUT

Code Trace: `qsort` and the Comparison Function



printf: a[0] = 15.

Step 7 / 11

```
1  int ascending(const void *p, const void *q) {
2      int x = *(const int *)p;
3      int y = *(const int *)q;
4      return (x > y) - (x < y);  // -1, 0, +1
5  }
6  int main(void) {
7      int a[] = {37, 15, 24, 33, 49};
8      size_t n = sizeof a / sizeof a[0];
9      qsort(a, n, sizeof a[0], ascending);
▶      for (size_t i = 0; i < n; i++)
▶          printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

`i = 0` `a = [15, 24, 33, 37, 49]`

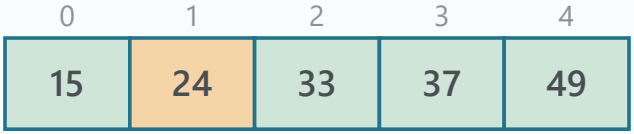
CALL STACK

`main()`

OUTPUT

```
15
```

Code Trace: `qsort` and the Comparison Function



printf: a[1] = 24.

Step 8 / 11

```
 1 int ascending(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 9     qsort(a, n, sizeof a[0], ascending);
▶       for (size_t i = 0; i < n; i++)
▶           printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

`i = 1` `a = [15, 24, 33, 37, 49]`

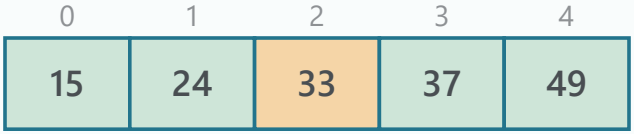
CALL STACK

`main()`

OUTPUT

```
15 24
```

Code Trace: `qsort` and the Comparison Function



printf: a[2] = 33.

Step 9 / 11

```
 1 int ascending(const void *p, const void *q) {
 2     int x = *(const int *)p;
 3     int y = *(const int *)q;
 4     return (x > y) - (x < y);  // -1, 0, +1
 5 }
 6 int main(void) {
 7     int a[] = {37, 15, 24, 33, 49};
 8     size_t n = sizeof a / sizeof a[0];
 9     qsort(a, n, sizeof a[0], ascending);
▶       for (size_t i = 0; i < n; i++)
▶           printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

`i = 2` `a = [15, 24, 33, 37, 49]`

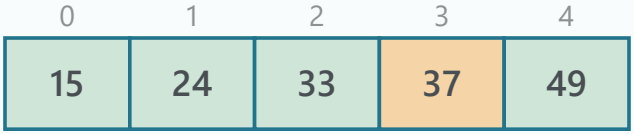
CALL STACK

`main()`

OUTPUT

```
15 24 33
```

Code Trace: `qsort` and the Comparison Function



printf: a[3] = 37.

Step 10 / 11

```
1  int ascending(const void *p, const void *q) {
2      int x = *(const int *)p;
3      int y = *(const int *)q;
4      return (x > y) - (x < y);  // -1, 0, +1
5  }
6  int main(void) {
7      int a[] = {37, 15, 24, 33, 49};
8      size_t n = sizeof a / sizeof a[0];
9      qsort(a, n, sizeof a[0], ascending);
▶      for (size_t i = 0; i < n; i++)
▶          printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

`i = 3` `a = [15, 24, 33, 37, 49]`

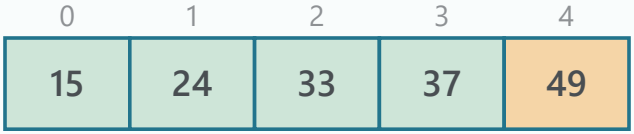
CALL STACK

`main()`

OUTPUT

```
15 24 33 37
```

Code Trace: `qsort` and the Comparison Function



printf: a[4] = 49.

Step 11 / 11

```
1  int ascending(const void *p, const void *q) {
2      int x = *(const int *)p;
3      int y = *(const int *)q;
4      return (x > y) - (x < y);  // -1, 0, +1
5  }
6  int main(void) {
7      int a[] = {37, 15, 24, 33, 49};
8      size_t n = sizeof a / sizeof a[0];
9      qsort(a, n, sizeof a[0], ascending);
▶      for (size_t i = 0; i < n; i++)
▶          printf("%d ", a[i]);
12     return 0;
13 }
```

VARIABLES

`i = 4` `a = [15, 24, 33, 37, 49]`

CALL STACK

`main()`

OUTPUT

```
15 24 33 37 49
```

Sorting Records (Structs) with qsort

```c
typedef struct { char name[20]; int grade; } student;

int by_name(const void *p, const void *q) {
    const student *x = p, *y = q;
    return strcmp(x->name, y->name);    // alphabetical
}
int by_grade_desc(const void *p, const void *q) {
    const student *x = p, *y = q;
    return (y->grade > x->grade) - (y->grade < x->grade);
}
/* qsort(s, n, sizeof s[0], by_name); */
```

The contract

Returns negative / zero / positive and must be **consistent** (a < b, b < c ⇒ a < c).

Trap: overflow

`return x - y;` is short but overflows for `INT_MIN` and `1`; `(x > y) - (x < y)` is safe.

No stability

qsort need not be stable or use any set algorithm; if equal keys' order matters, add a secondary key.

Hybrid Sorts: Combining Algorithms

Quicksort + insertion

On small pieces the cost of recursion dominates; pieces below M are finished by insertion sort (quicksort3 + sort).

Introsort (Musser, 1997)

Starts with quicksort; if the recursion depth exceeds ≈ 2⌊lg n⌋, it switches to **heap sort** on that piece → worst case O(n lg n). Small pieces go to insertion sort. Common in C++ `std::sort` implementations.

Timsort (Peters, 2002)

Merge + insertion sort; finds already sorted *runs* in the data and merges them. **Stable**; Python's `sort` and Java's `Arrays.sort` for object arrays.

The common idea

Good asymptotic behavior at scale (n lg n) + a low constant on small inputs (insertion) + insurance against the worst case.

Merge Sort

1. A list of 0 or 1 elements is already sorted.
2. Otherwise split the list into two halves as equal as possible.
3. Sort each half by applying merge sort recursively.
4. **Merge** the two sorted sublists into one sorted list.

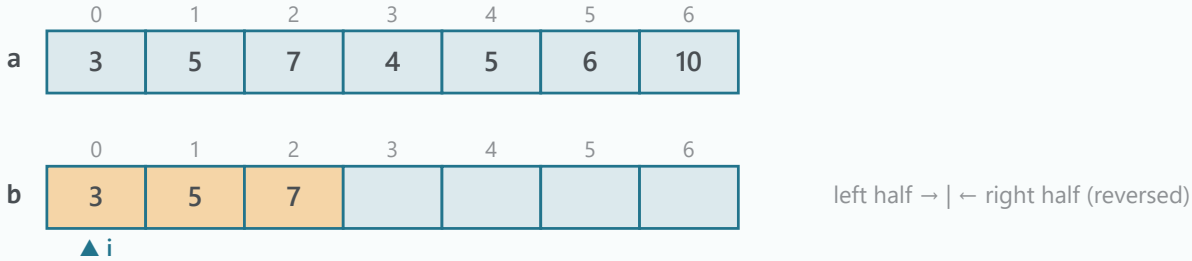
- **John von Neumann**, 1945; divide and conquer.
- Idea 1: sorting a small list takes fewer steps.
- Idea 2: building a sorted list from two sorted lists is cheaper than from two unsorted ones.

```
void mergesort(int a[], int b[], int L, int R) {
    if (R <= L) return;
    int m = (R + L) / 2;
    mergesort(a, b, L, m);
    mergesort(a, b, m + 1, R);
    merge(a, b, L, m, R);
}
```

```
void merge(int a[], int b[], int L, int m, int R) {
    int i, j, k;
    for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
    for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
    for (k = L; k <= R; k++)
        if (less(b[j], b[i]))
            a[k] = b[j--];
        else
            a[k] = b[i++];
}
```

The auxiliary array `b[]` (same size as a) is passed as a parameter; it cannot be used without being defined.

Code Trace: `merge(a, b, 0, 2, 6)`



Loop 1: the left half a[0..2] is copied as is to b[0..2]; the loop exits with **i = L = 0**.

Step 1 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
▶      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
6          if (less(b[j], b[i]))
7              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

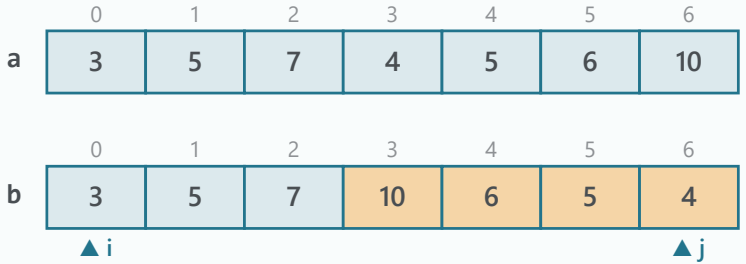
VARIABLES

`i = 0` `j = ?` `k = ?`

`a = [3, 5, 7, 4, 5, 6, 10]`

`b = [3, 5, 7, _, _, _, _]`

Code Trace: `merge(a, b, 0, 2, 6)`



Loop 2: the right half is copied to b[3..6] **in reverse** → b increases from both ends toward the middle (bitonic). The loop exits with **j = R = 6**.

Step 2 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
▶      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
6          if (less(b[j], b[i]))
7              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

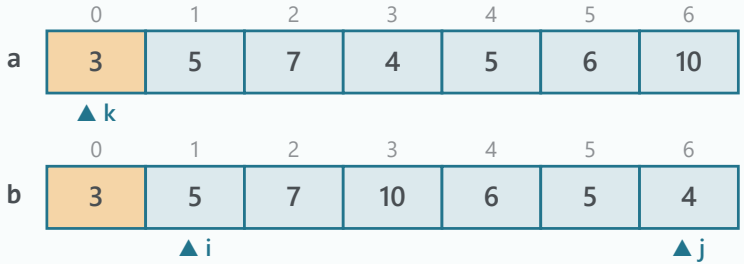
VARIABLES

`i = 0` `j = 6` `k = ?`

`a = [3, 5, 7, 4, 5, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Code Trace: `merge(a, b, 0, 2, 6)`



left half → | ← right half (reversed)

b[j] = 4 ≥ b[i] = 3 → a[0] = b[0]; i++.

Step 3 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (less(b[j], b[i]))
7              a[k] = b[j--];
8          else
▶              a[k] = b[i++];
10 }
```

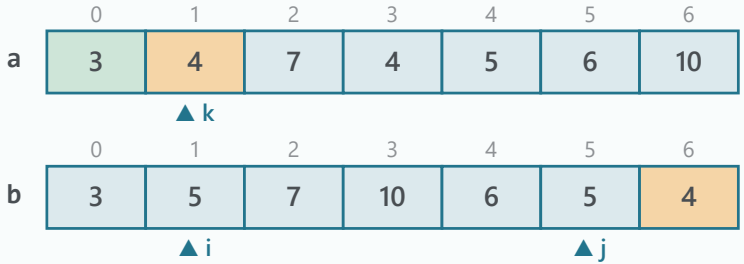
VARIABLES

`i = 1` `j = 6` `k = 0`

`a = [3, 5, 7, 4, 5, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Code Trace: `merge(a, b, 0, 2, 6)`



left half → | ← right half (reversed)

b[j] = 4 < b[i] = 5 → a[1] = b[6]; j--.

Step 4 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (less(b[j], b[i]))
▶              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

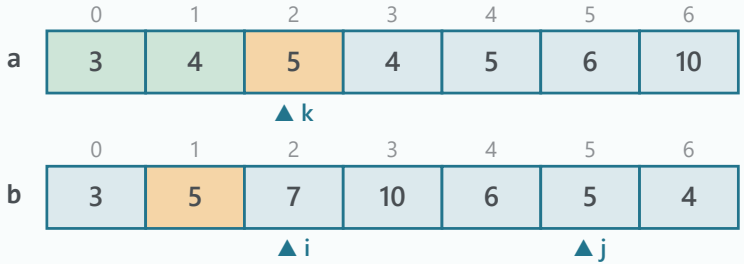
VARIABLES

`i = 1` `j = 5` `k = 1`

`a = [3, 4, 7, 4, 5, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Code Trace: `merge(a, b, 0, 2, 6)`



left half → | ← right half (reversed)

b[j] = 5 ≥ b[i] = 5 → a[2] = b[1]; i++.

Step 5 / 10

```
 1 void merge(int a[], int b[], int L, int m, int R) {
 2     int i, j, k;
 3     for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
 4     for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
 5     for (k = L; k <= R; k++)
▶            if (less(b[j], b[i]))
 7               a[k] = b[j--];
 8           else
▶               a[k] = b[i++];
10 }
```

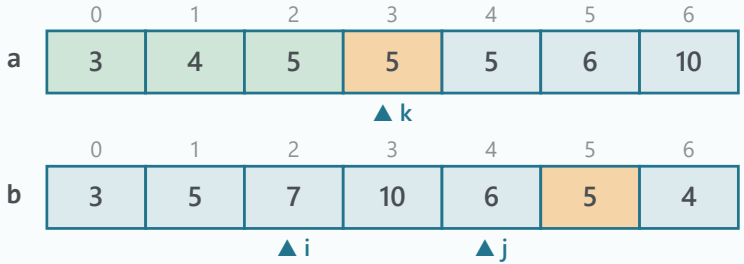
VARIABLES

`i = 2` `j = 5` `k = 2`

`a = [3, 4, 5, 4, 5, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Code Trace: `merge(a, b, 0, 2, 6)`



left half → | ← right half (reversed)

b[j] = 5 < b[i] = 7 → a[3] = b[5]; j--.

Step 6 / 10

```
 1 void merge(int a[], int b[], int L, int m, int R) {
 2     int i, j, k;
 3     for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
 4     for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
 5     for (k = L; k <= R; k++)
▶            if (less(b[j], b[i]))
▶                a[k] = b[j--];
 8         else
 9             a[k] = b[i++];
10 }
```

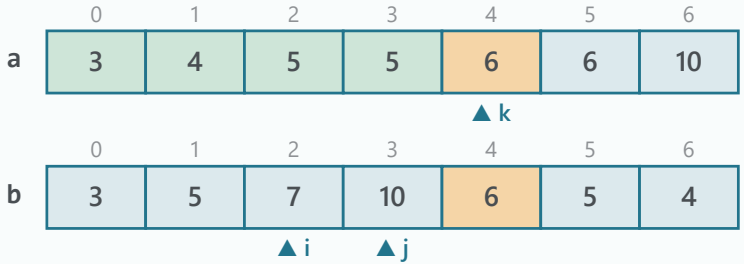
VARIABLES

`i = 2` `j = 4` `k = 3`

`a = [3, 4, 5, 5, 5, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Code Trace: `merge(a, b, 0, 2, 6)`



left half → | ← right half (reversed)

b[j] = 6 < b[i] = 7 → a[4] = b[4]; j--.

Step 7 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (less(b[j], b[i]))
▶              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

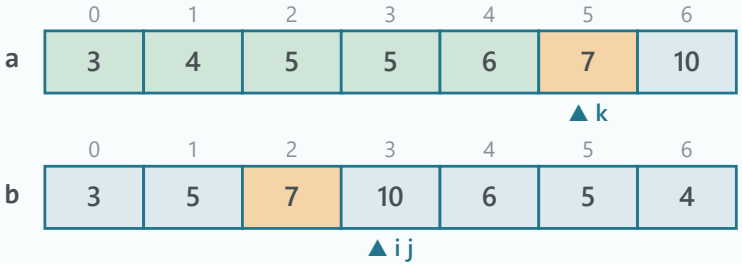
VARIABLES

`i = 2` `j = 3` `k = 4`

`a = [3, 4, 5, 5, 6, 6, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Code Trace: `merge(a, b, 0, 2, 6)`



left half → | ← right half (reversed)

b[j] = 10 ≥ b[i] = 7 → a[5] = b[2]; i++.

Step 8 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (less(b[j], b[i]))
7              a[k] = b[j--];
8          else
▶              a[k] = b[i++];
10 }
```

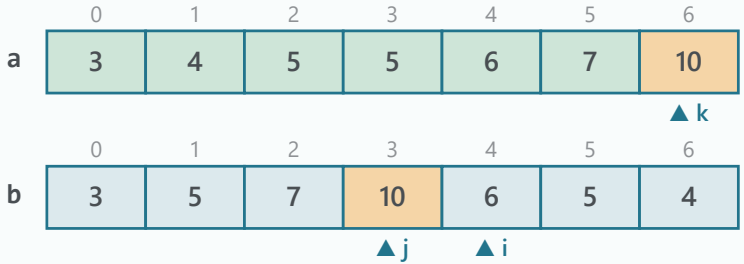
VARIABLES

`i = 3` `j = 3` `k = 5`

`a = [3, 4, 5, 5, 6, 7, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Code Trace: `merge(a, b, 0, 2, 6)`



left half → | ← right half (reversed)

b[j] = 10 ≥ b[i] = 10 → a[6] = b[3]; i++.

Step 9 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
5      for (k = L; k <= R; k++)
▶          if (less(b[j], b[i]))
7              a[k] = b[j--];
8          else
▶              a[k] = b[i++];
10 }
```

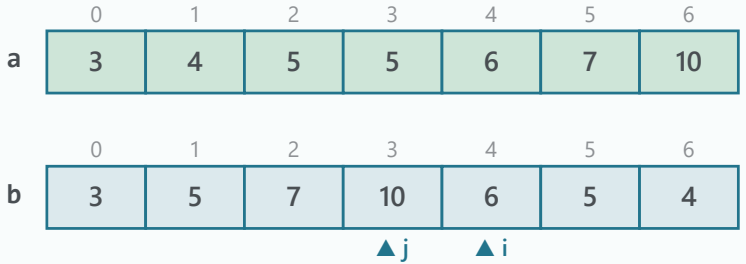
VARIABLES

`i = 4` `j = 3` `k = 6`

`a = [3, 4, 5, 5, 6, 7, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Code Trace: `merge(a, b, 0, 2, 6)`



Done: a = [3, 4, 5, 5, 6, 7, 10]. At the end i = 4, j = 3: the two indices met in the middle; no bounds test was needed. Of the equal 5s, the left one (b[i]) was taken first.

Step 10 / 10

```
1  void merge(int a[], int b[], int L, int m, int R) {
2      int i, j, k;
3      for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
4      for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
▶      for (k = L; k <= R; k++)
6          if (less(b[j], b[i]))
7              a[k] = b[j--];
8          else
9              a[k] = b[i++];
10 }
```

VARIABLES

`i = 4` `j = 3` `k = 6`

`a = [3, 4, 5, 5, 6, 7, 10]`

`b = [3, 5, 7, 10, 6, 5, 4]`

Code Trace: `mergesort`, 9 Merge Calls

0	1	2	3	4	5	6	7	8	9
89	67	59	62	23	93	24	20	28	60

mergesort(a, b, 0, 9): the left half is sorted completely first; the first merge happens only in the deepest call, (0, 1).

Step 1 / 10

```
▶ void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
6     merge(a, b, L, m, R);
7 }
```

VARIABLES

L = 0 R = 9

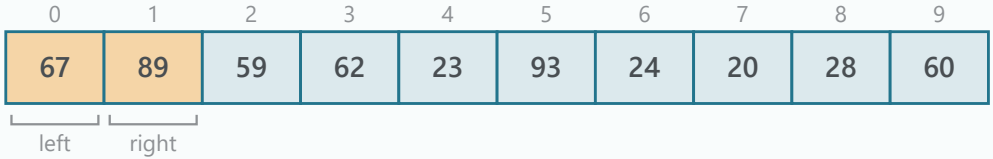
a = [89, 67, 59, 62, 23, 93, 24, 20, 28, 60]

CALL STACK

mergesort(0, 9)

main()

Code Trace: `mergesort`, 9 Merge Calls



merge #1: a[0..0] and a[1..1] merged → a[0..1] sorted.

Step 2 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

VARIABLES

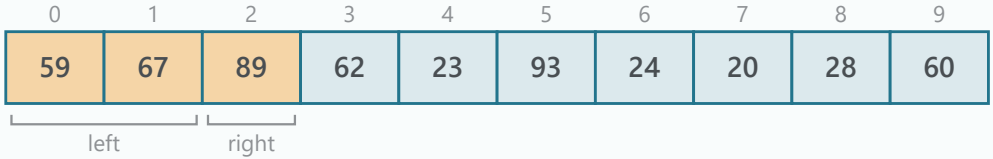
`L = 0` `m = 0` `R = 1`

`a = [67, 89, 59, 62, 23, 93, 24, 20, 28, 60]`

CALL STACK

- `merge(0, 0, 1)`
- `mergesort(0, 1)`
- `mergesort(0, 2)`
- `mergesort(0, 4)`
- `mergesort(0, 9)`
- `main()`

Code Trace: `mergesort`, 9 Merge Calls



merge #2: a[0..1] and a[2..2] merged → a[0..2] sorted.

Step 3 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

VARIABLES

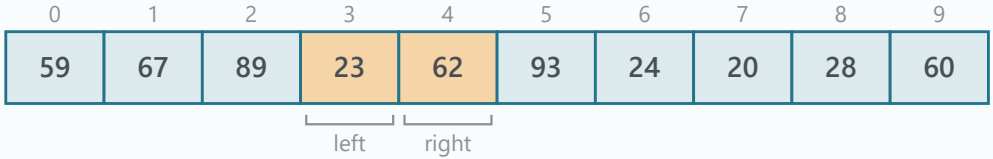
`L = 0` `m = 1` `R = 2`

`a = [59, 67, 89, 62, 23, 93, 24, 20, 28, 60]`

CALL STACK

- `merge(0, 1, 2)`
- `mergesort(0, 2)`
- `mergesort(0, 4)`
- `mergesort(0, 9)`
- `main()`

Code Trace: `mergesort`, 9 Merge Calls



merge #3: a[3..3] and a[4..4] merged → a[3..4] sorted.

Step 4 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

VARIABLES

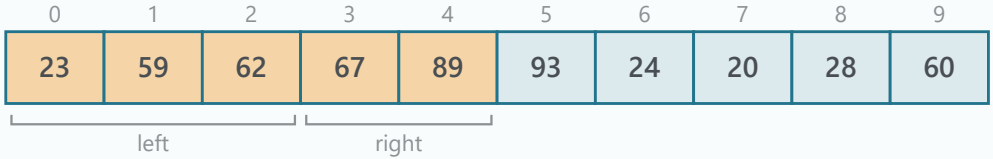
`L = 3` `m = 3` `R = 4`

`a = [59, 67, 89, 23, 62, 93, 24, 20, 28, 60]`

CALL STACK

- `merge(3, 3, 4)`
- `mergesort(3, 4)`
- `mergesort(0, 4)`
- `mergesort(0, 9)`
- `main()`

Code Trace: `mergesort`, 9 Merge Calls



merge #4: a[0..2] and a[3..4] merged → a[0..4] sorted.

Step 5 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

VARIABLES

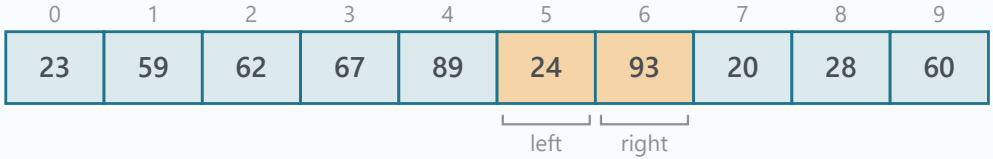
`L = 0` `m = 2` `R = 4`

`a = [23, 59, 62, 67, 89, 93, 24, 20, 28, 60]`

CALL STACK

- `merge(0, 2, 4)`
- `mergesort(0, 4)`
- `mergesort(0, 9)`
- `main()`

Code Trace: `mergesort`, 9 Merge Calls



merge #5: a[5..5] and a[6..6] merged → a[5..6] sorted.

Step 6 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

VARIABLES

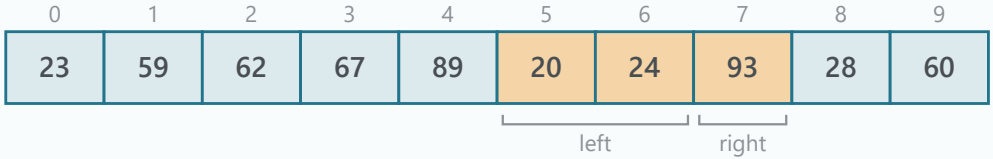
`L = 5` `m = 5` `R = 6`

`a = [23, 59, 62, 67, 89, 24, 93, 20, 28, 60]`

CALL STACK

- **`merge(5, 5, 6)`**
- `mergesort(5, 6)`
- `mergesort(5, 7)`
- `mergesort(5, 9)`
- `mergesort(0, 9)`
- `main()`

Code Trace: `mergesort`, 9 Merge Calls



merge #6: a[5..6] and a[7..7] merged → a[5..7] sorted.

Step 7 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

VARIABLES

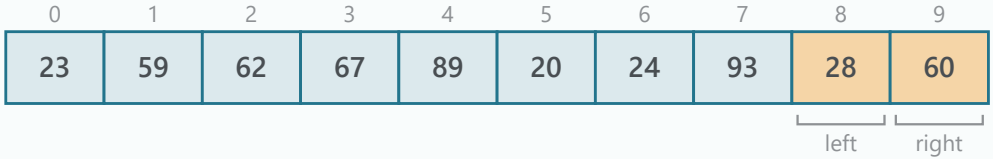
`L = 5` `m = 6` `R = 7`

`a = [23, 59, 62, 67, 89, 20, 24, 93, 28, 60]`

CALL STACK

- `merge(5, 6, 7)`
- `mergesort(5, 7)`
- `mergesort(5, 9)`
- `mergesort(0, 9)`
- `main()`

Code Trace: `mergesort`, 9 Merge Calls



merge #7: a[8..8] and a[9..9] merged → a[8..9] sorted.

Step 8 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

VARIABLES

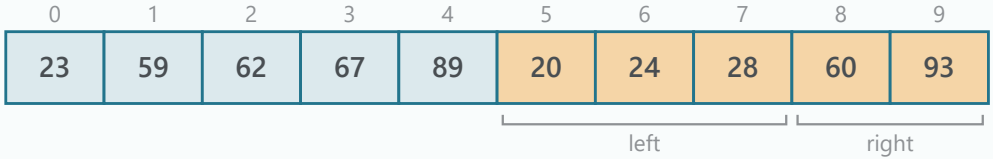
`L = 8` `m = 8` `R = 9`

`a = [23, 59, 62, 67, 89, 20, 24, 93, 28, 60]`

CALL STACK

- `merge(8, 8, 9)`
- `mergesort(8, 9)`
- `mergesort(5, 9)`
- `mergesort(0, 9)`
- `main()`

Code Trace: `mergesort`, 9 Merge Calls



merge #8: a[5..7] and a[8..9] merged → a[5..9] sorted.

Step 9 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

VARIABLES

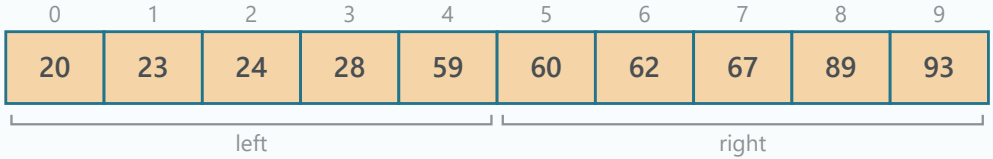
`L = 5` `m = 7` `R = 9`

`a = [23, 59, 62, 67, 89, 20, 24, 28, 60, 93]`

CALL STACK

- `merge(5, 7, 9)`
- `mergesort(5, 9)`
- `mergesort(0, 9)`
- `main()`

Code Trace: `mergesort`, 9 Merge Calls



merge #9: a[0..4] and a[5..9] merged → a[0..9] sorted.

Step 10 / 10

```
1 void mergesort(int a[], int b[], int L, int R) {
2     if (R <= L) return;
3     int m = (R + L) / 2;
4     mergesort(a, b, L, m);
5     mergesort(a, b, m + 1, R);
▶     merge(a, b, L, m, R);
7 }
```

VARIABLES

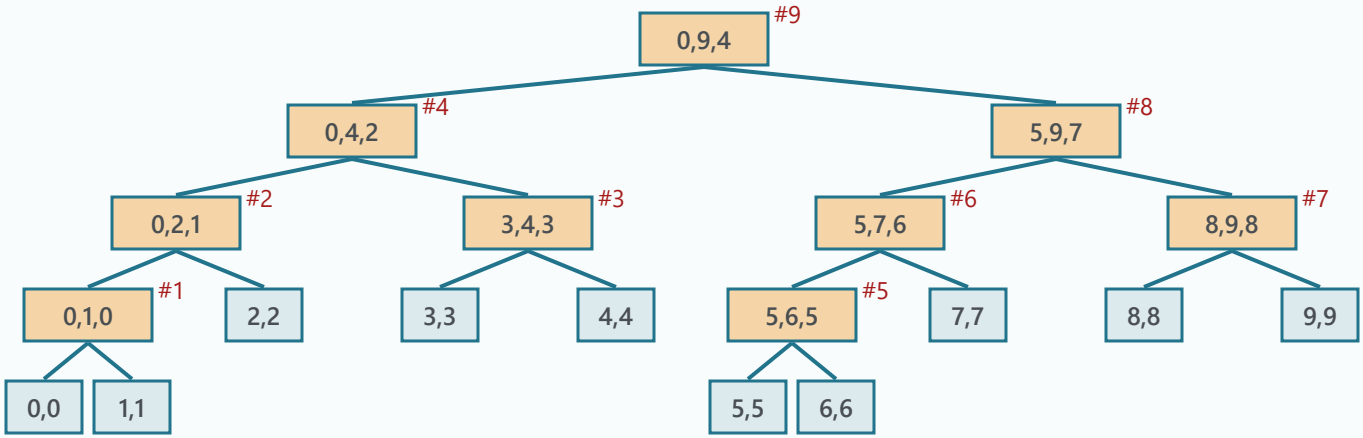
`L = 0` `m = 4` `R = 9`

`a = [20, 23, 24, 28, 59, 60, 62, 67, 89, 93]`

CALL STACK

- `merge(0, 4, 9)`
- `mergesort(0, 9)`
- `main()`

The Tree of Stack Frames



Box: L, R, m · orange: a frame that calls merge, #k: which merge. It goes deep on the left first and merges on the way back.

INSTRUCTOR “In Data Structures 1 I always said: ‘We call this logic merge.’”

Question: What If the Two mergesort Lines Swap?

Question: Swap the two `mergesort` lines in the function and observe the table you get.

▶ **Answer: the right halves merge first; same result, different intermediate tables**

Merge Sort: Time and Space Complexity

$$a_n = a_{\lceil n/2 \rceil} + a_{\lfloor n/2 \rfloor} + n, \quad a_1 = 0$$

$$n = 2^k: a_n = 2a_{n/2} + n \Rightarrow a_n = \Theta(n \lg n)$$

- 16 elements split into 8 + 8, 17 into 8 + 9; merge takes n steps.
- Number of calls $T_n = 2T_{n/2} + 1 \Rightarrow 2n - 1$ (19 calls for 10 elements, 9 of which merge).
- **Independent** of the input order: best = average = worst = n lg n.
- Drawback: **O(n) extra memory**. Avoiding it raises the cost; if no extra memory is allowed, heap sort is preferred.
- Because it merges sorted runs in sequence, it is the basis of **external sorting** on disk.

Stability

The classic merge (taking the left one on ties) is **stable** (quicksort and heap sort are not). The reverse-copy (bitonic) merge here is not stable: [2a, 2b, 2c, 2d] → [2a, 2b, 2d, 2c] (checked in Python).

INSTRUCTOR "Since splitting the input in half is the clever part, this one splits it in half too. But in the earlier ones the pivot landed in its right place in one step; here nothing does. It builds a new sorted array by exploiting the fact that the two arrays are already sorted."

Sorting Alphabetic Data: Linked Merge Sort

```
typedef struct {
    int key;        /* + large data fields */
    int link;       /* index of the next item, 0 = end */
} element;

int mergesort(element a[], int L, int R) {
    if (R <= L) return L;           /* one-item list */
    int m = (R + L) / 2;
    int right = mergesort(a, m + 1, R);
    int left = mergesort(a, L, m);
    return merge(a, left, right);  /* head index */
}
```

merge() links two sorted lists through their link fields and returns the head index.

- Copying large records (e.g. alphabetic data) is slow. The fix: build a linked list inside the array with a **link** field; only indices change.
- The data is in a[1..n]; **a[0]** is a temporary head during merge. Initially every link is 0.
- mergesort returns the **head index** of the sorted sublist.
- **Common mistakes:** returning a value from a `void` function; i1 instead of i2 in the `else` branch; returning `a[c].link` instead of a[0].
- In C the order in which arguments are evaluated is **unspecified**; here the order is written out explicitly.

Code Trace: Linked `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
key	–	71	77	122	38	119	41	22	115	43	13
link	4	2	5	0	1	3	9	6	0	8	7
	▲ last				▲ i1						▲ i2

merge #9: first list (head 4) 38→71→77→119→122; second list (head 10) 13→22→41→43→115. last = 0 (a[0] is the temporary head).

Step 1 / 10

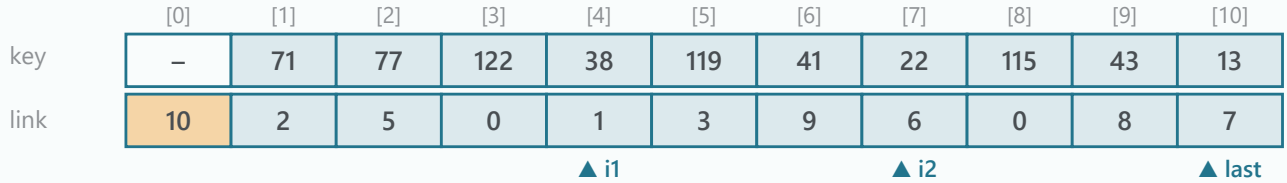
```
1  int merge(element a[], int first, int second) {
▶      int last = 0;               // a[0]: temp head
▶      int i1 = first, i2 = second;
4      while (i1 && i2)
5          if (a[i1].key <= a[i2].key) {
6              a[last].link = i1; last = i1;
7              i1 = a[i1].link;
8          } else {
9              a[last].link = i2; last = i2;
10             i2 = a[i2].link;
11         }
12     a[last].link = i1 ? i1 : i2;
13     return a[0].link;
14 }
```

VARIABLES

`i1 = 4` `i2 = 10` `last = 0`

`link = [4, 2, 5, 0, 1, 3, 9, 6, 0, 8, 7]`

Code Trace: Linked `merge` (#9)



38 > 13 → a[0].link = 10; last = 10; i2 = 7.

Step 2 / 10

```
1  int merge(element a[], int first, int second) {
2      int last = 0;                // a[0]: temp head
3      int i1 = first, i2 = second;
4      while (i1 && i2)
5          if (a[i1].key <= a[i2].key) {
6              a[last].link = i1; last = i1;
7              i1 = a[i1].link;
▶          } else {
▶              a[last].link = i2; last = i2;
▶              i2 = a[i2].link;
11         }
12     a[last].link = i1 ? i1 : i2;
13     return a[0].link;
14 }
```

VARIABLES

`i1 = 4` `i2 = 7` `last = 10`

`link = [10, 2, 5, 0, 1, 3, 9, 6, 0, 8, 7]`

Code Trace: Linked `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
key	–	71	77	122	38	119	41	22	115	43	13
link	10	2	5	0	1	3	9	6	0	8	7
					▲ i1		▲ i2	▲ last			

38 > 22 → a[10].link = 7; last = 7; i2 = 6.

Step 3 / 10

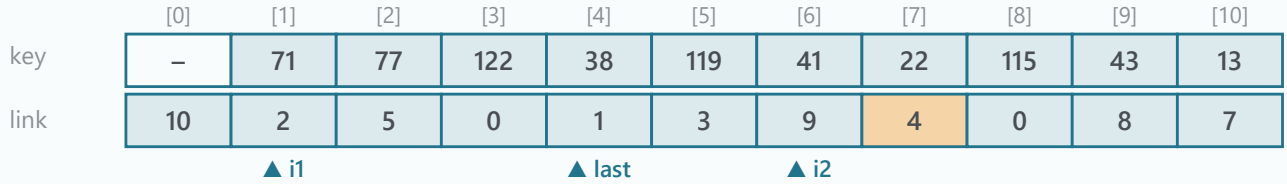
```
1  int merge(element a[], int first, int second) {
2      int last = 0;                // a[0]: temp head
3      int i1 = first, i2 = second;
4      while (i1 && i2)
5          if (a[i1].key <= a[i2].key) {
6              a[last].link = i1; last = i1;
7              i1 = a[i1].link;
▶          } else {
▶              a[last].link = i2; last = i2;
▶              i2 = a[i2].link;
11         }
12     a[last].link = i1 ? i1 : i2;
13     return a[0].link;
14 }
```

VARIABLES

`i1 = 4` `i2 = 6` `last = 7`

`link = [10, 2, 5, 0, 1, 3, 9, 6, 0, 8, 7]`

Code Trace: Linked `merge` (#9)



38 ≤ 41 → a[7].link = 4; last = 4; i1 = 1.

Step 4 / 10

```
1  int merge(element a[], int first, int second) {
2      int last = 0;                // a[0]: temp head
3      int i1 = first, i2 = second;
4      while (i1 && i2)
▶          if (a[i1].key <= a[i2].key) {
▶              a[last].link = i1; last = i1;
▶              i1 = a[i1].link;
8          } else {
9              a[last].link = i2; last = i2;
10             i2 = a[i2].link;
11         }
12     a[last].link = i1 ? i1 : i2;
13     return a[0].link;
14 }
```

VARIABLES

`i1 = 1` `i2 = 6` `last = 4`

`link = [10, 2, 5, 0, 1, 3, 9, 4, 0, 8, 7]`

Code Trace: Linked `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
key	–	71	77	122	38	119	41	22	115	43	13
link	10	2	5	0	6	3	9	4	0	8	7
		▲ i1					▲ last			▲ i2	

71 > 41 → a[4].link = 6; last = 6; i2 = 9.

Step 5 / 10

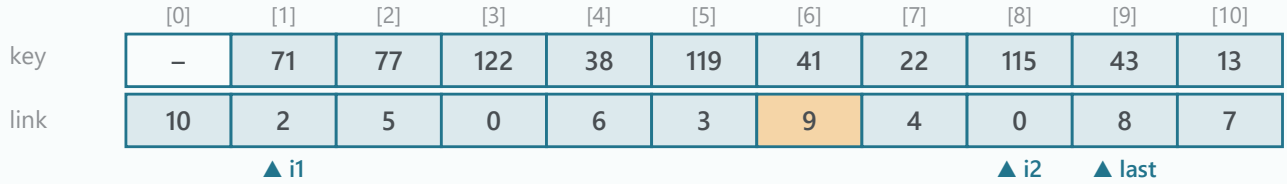
```
1  int merge(element a[], int first, int second) {
2      int last = 0;                // a[0]: temp head
3      int i1 = first, i2 = second;
4      while (i1 && i2)
5          if (a[i1].key <= a[i2].key) {
6              a[last].link = i1; last = i1;
7              i1 = a[i1].link;
▶          } else {
▶              a[last].link = i2; last = i2;
▶              i2 = a[i2].link;
11         }
12     a[last].link = i1 ? i1 : i2;
13     return a[0].link;
14 }
```

VARIABLES

`i1 = 1` `i2 = 9` `last = 6`

`link = [10, 2, 5, 0, 6, 3, 9, 4, 0, 8, 7]`

Code Trace: Linked `merge` (#9)



71 > 43 → a[6].link = 9; last = 9; i2 = 8.

Step 6 / 10

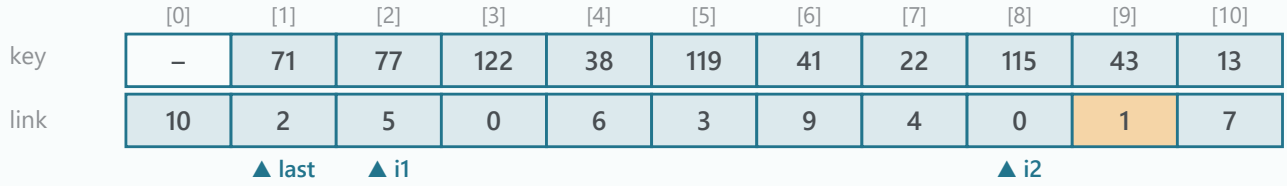
```
1  int merge(element a[], int first, int second) {
2      int last = 0;                // a[0]: temp head
3      int i1 = first, i2 = second;
4      while (i1 && i2)
5          if (a[i1].key <= a[i2].key) {
6              a[last].link = i1; last = i1;
7              i1 = a[i1].link;
▶          } else {
▶              a[last].link = i2; last = i2;
▶              i2 = a[i2].link;
11         }
12     a[last].link = i1 ? i1 : i2;
13     return a[0].link;
14 }
```

VARIABLES

`i1 = 1` `i2 = 8` `last = 9`

`link = [10, 2, 5, 0, 6, 3, 9, 4, 0, 8, 7]`

Code Trace: Linked `merge` (#9)



71 ≤ 115 → a[9].link = 1; last = 1; i1 = 2.

Step 7 / 10

```
1  int merge(element a[], int first, int second) {
2      int last = 0;                // a[0]: temp head
3      int i1 = first, i2 = second;
4      while (i1 && i2)
▶          if (a[i1].key <= a[i2].key) {
▶              a[last].link = i1; last = i1;
▶              i1 = a[i1].link;
8          } else {
9              a[last].link = i2; last = i2;
10             i2 = a[i2].link;
11         }
12     a[last].link = i1 ? i1 : i2;
13     return a[0].link;
14 }
```

VARIABLES

`i1 = 2` `i2 = 8` `last = 1`

`link = [10, 2, 5, 0, 6, 3, 9, 4, 0, 1, 7]`

Code Trace: Linked `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
key	–	71	77	122	38	119	41	22	115	43	13
link	10	2	5	0	6	3	9	4	0	1	7
			▲ last			▲ i1			▲ i2		

77 ≤ 115 → a[1].link = 2; last = 2; i1 = 5.

Step 8 / 10

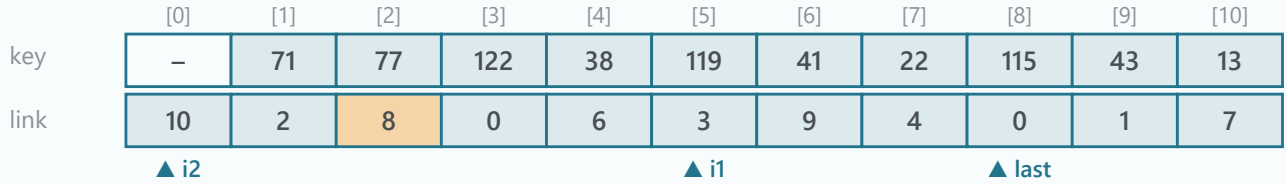
```
1  int merge(element a[], int first, int second) {
2      int last = 0;                // a[0]: temp head
3      int i1 = first, i2 = second;
4      while (i1 && i2)
▶          if (a[i1].key <= a[i2].key) {
▶              a[last].link = i1; last = i1;
▶              i1 = a[i1].link;
8          } else {
9              a[last].link = i2; last = i2;
10             i2 = a[i2].link;
11         }
12     a[last].link = i1 ? i1 : i2;
13     return a[0].link;
14 }
```

VARIABLES

`i1 = 5` `i2 = 8` `last = 2`

`link = [10, 2, 5, 0, 6, 3, 9, 4, 0, 1, 7]`

Code Trace: Linked `merge` (#9)



119 > 115 → a[2].link = 8; last = 8; i2 = 0.

Step 9 / 10

```
1  int merge(element a[], int first, int second) {
2      int last = 0;                // a[0]: temp head
3      int i1 = first, i2 = second;
4      while (i1 && i2)
5          if (a[i1].key <= a[i2].key) {
6              a[last].link = i1; last = i1;
7              i1 = a[i1].link;
▶          } else {
▶              a[last].link = i2; last = i2;
▶              i2 = a[i2].link;
11         }
12     a[last].link = i1 ? i1 : i2;
13     return a[0].link;
14 }
```

VARIABLES

`i1 = 5` `i2 = 0` `last = 8`

`link = [10, 2, 8, 0, 6, 3, 9, 4, 0, 1, 7]`

Code Trace: Linked `merge` (#9)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
key	–	71	77	122	38	119	41	22	115	43	13
link	10	2	8	0	6	3	9	4	5	1	7
									▲ last		

i2 = 0 (the second list is done) → a[8].link = i1 = 5; the head **a[0].link = 10** is returned: 13→22→38→41→43→71→77→115→119→122.

Step 10 / 10

```
1  int merge(element a[], int first, int second) {
2      int last = 0;               // a[0]: temp head
3      int i1 = first, i2 = second;
4      while (i1 && i2)
5          if (a[i1].key <= a[i2].key) {
6              a[last].link = i1; last = i1;
7              i1 = a[i1].link;
8          } else {
9              a[last].link = i2; last = i2;
10             i2 = a[i2].link;
11         }
▶      a[last].link = i1 ? i1 : i2;
▶      return a[0].link;
14 }
```

VARIABLES

`i1 = 5` `i2 = 0` `last = 8`

`link = [10, 2, 8, 0, 6, 3, 9, 4, 5, 1, 7]`

Linked Merge Sort: How the Link Fields Change

#	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	returns	L, R
key	71	77	122	38	119	41	22	115	43	13		
#1	0	0	0	0	0	0	0	0	0	**9**	10	9,10
#2	0	0	0	0	0	0	**6**	0	0	9	7	6,7
#3	0	0	0	0	0	**8**	6	0	0	9	7	6,8
#4	0	0	0	0	0	**9**	6	0	**8**	**7**	10	6,10
#5	0	0	0	**5**	0	9	6	0	8	7	4	4,5
#6	**2**	0	0	5	0	9	6	0	8	7	1	1,2
#7	2	**3**	0	5	0	9	6	0	8	7	1	1,3
#8	2	**5**	0	**1**	**3**	9	6	0	8	7	4	1,5
#9	2	**8**	0	**6**	3	9	**4**	**5**	**1**	7	10	1,10

Bold: the links changed by that merge. Sorted list: 13 → 22 → 38 → 41 → 43 → 71 → 77 → 115 → 119 → 122.

Exam Question

The adaptation below sorts the `element` array a[1..9] by key. (i) Draw the **stack frames** of the recursive calls; where merge is called, write its return value. (ii) Fill in the execution table with merge numbers, return values and the link fields at each step.

```
int mergesort(element a[], int L, int R) {
    int m;
    if (R <= L) return L;
    m = (R + L) / 2;
    return merge(a, mergesort(a, m + 1, R),
                    mergesort(a, L, m));
}
```

Keys: [1] ?, [2] ?, [3] 18, [4] −5, [5] 27, [6] 11, [7] ?, [8] 83, [9] 30. In place of the "?" on the exam sheet we use 42, 7, 55. merge() is as in linked merge sort; the printed version has i1 in the `else` branch (a typo; it should be i2).

Exam Question (Question 3): Solution

▼ **Solution**

If the arguments are evaluated left to right (right half first), the call order is: (1,9) → (6,9) → (8,9) → (9,9) → (8,8) → (6,7) → (7,7) → (6,6) → (1,5) …

#	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	ret.	L,R,m
#1	0	0	0	0	0	0	0	0	8	9	8,9,8
#2	0	0	0	0	0	7	0	0	8	6	6,7,6
#3	0	0	0	0	0	9	8	0	7	6	6,9,7
#4	0	0	0	5	0	9	8	0	7	4	4,5,4
#5	0	1	0	5	0	9	8	0	7	2	1,2,1
#6	0	3	1	5	0	9	8	0	7	2	1,3,2
#7	0	3	5	2	1	9	8	0	7	4	1,5,3
#8	7	6	5	2	9	3	8	0	1	4	1,9,5

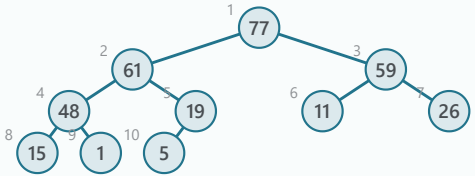
Resulting list (head 4): -5 → 7 → 11 → 18 → 27 → 30 → 42 → 55 → 83. The table was produced by simulation.

Heap Sort

```c
void heapify(int a[], int root, int n) {
    int t = a[root], j;
    for (j = 2 * root; j <= n; j *= 2) {
        if (j < n && a[j] < a[j + 1]) j++; // larger
        if (t >= a[j]) break;   // t fits here
        a[j / 2] = a[j];        // move child up
    }
    a[j / 2] = t;
}
void heapsort(int a[], int n) {     /* a[1..n] */
    int i;
    for (i = n / 2; i >= 1; i--)     /* build heap */
        heapify(a, i, n);
    for (i = n - 1; i >= 1; i--) {   /* sort */
        SWAP(a[1], a[i + 1]);    /* largest to the end */
        heapify(a, 1, i);            /* fix the heap */
    }
}
```

- **O(n lg n)** like merge sort, but it uses **no extra memory** (in place).
- The array a[1..n] (index 0 unused) is viewed as a **complete binary tree**: the children of a[i] are a[2i] and a[2i+1].
- `heapify(a, root, n)`: with the left and right subtrees of root already max heaps, finds the right place for the root value.
- Leaves are skipped: heapify nodes ⌊n/2⌋, ⌊n/2⌋−1, ..., 1 from the back → a max heap.
- Then the largest at the root moves to the end and the heap is fixed for the remaining i elements.
- A `j` or `i` declared in a `for` header cannot be used after the loop (out of scope in C99).

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
77	61	59	48	19	11	26	15	1	5

heap region a[1..10]

The input [26, 5, 77, 1, 61, 11, 59, 15, 48, 19] (Horowitz's example) was turned bottom-up into this max heap by heapify for i = 5...1 (5 moves in all). “Fine, we got that, but when does the array get sorted?” Now: each round the largest (77) at the root moves to the end and the rest of the heap is fixed.

Step 1 / 19

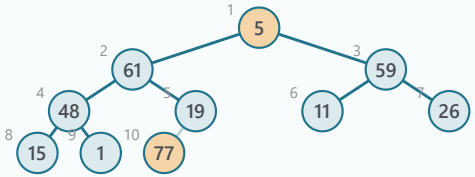
```
 1 void heapify(int a[], int root, int n) {
 2     int t = a[root], j;
 3     for (j = 2 * root; j <= n; j *= 2) {
 4         if (j < n && a[j] < a[j + 1]) j++; // larger
 5         if (t >= a[j]) break;   // t fits here
 6         a[j / 2] = a[j];        // move child up
 7     }
 8     a[j / 2] = t;
 9 }
▶    for (i = n - 1; i >= 1; i--) {  /* sort */
11         SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

VARIABLES

n = 10

a = [77, 61, 59, 48, 19, 11, 26, 15, 1, 5]

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
5	61	59	48	19	11	26	15	1	77

heap region a[1..9]

i = 9: SWAP(a[1], a[10]) → the largest, 77, goes to the end; 5 at the root breaks the heap.

Step 2 / 19

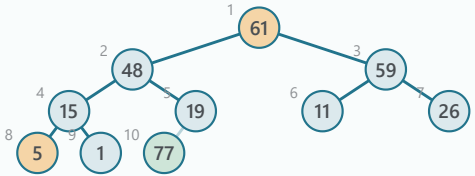
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

VARIABLES

`i = 9`

`a = [5, 61, 59, 48, 19, 11, 26, 15, 1, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
61	48	59	15	19	11	26	5	1	77

heap region a[1..9] · sorted part green

heapify(a, 1, 9): 5 sank to a[8]; the new root **61** is the largest remaining.

Step 3 / 19

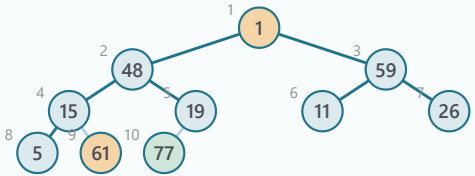
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

VARIABLES

`i = 9`

`a = [61, 48, 59, 15, 19, 11, 26, 5, 1, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	48	59	15	19	11	26	5	61	77

heap region a[1..8] · sorted part green

i = 8: SWAP(a[1], a[9]) → the largest, 61, goes to the end; 1 at the root breaks the heap.

Step 4 / 19

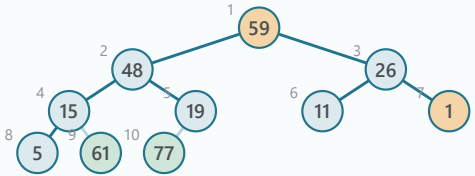
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

VARIABLES

`i = 8`

`a = [1, 48, 59, 15, 19, 11, 26, 5, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
59	48	26	15	19	11	1	5	61	77

heap region a[1..8] · sorted part green

heapify(a, 1, 8): 1 sank to a[7]; the new root **59** is the largest remaining.

Step 5 / 19

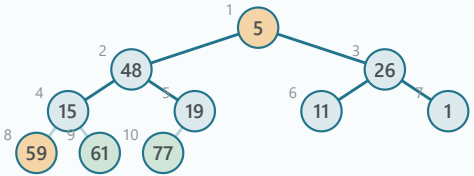
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

VARIABLES

`i = 8`

`a = [59, 48, 26, 15, 19, 11, 1, 5, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
5	48	26	15	19	11	1	59	61	77

heap region a[1..7] · sorted part green

i = 7: SWAP(a[1], a[8]) → the largest, 59, goes to the end; 5 at the root breaks the heap.

Step 6 / 19

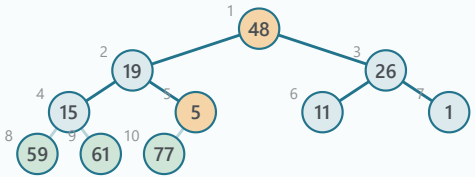
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

VARIABLES

`i = 7`

`a = [5, 48, 26, 15, 19, 11, 1, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
48	19	26	15	5	11	1	59	61	77

heap region a[1..7] · sorted part green

heapify(a, 1, 7): 5 sank to a[5]; the new root **48** is the largest remaining.

Step 7 / 19

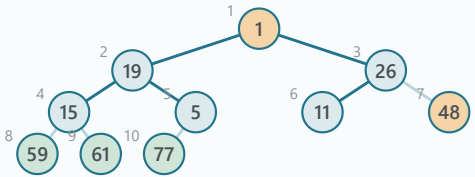
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

VARIABLES

`i = 7`

`a = [48, 19, 26, 15, 5, 11, 1, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	19	26	15	5	11	48	59	61	77

heap region a[1..6] · sorted part green

i = 6: SWAP(a[1], a[7]) → the largest, 48, goes to the end; 1 at the root breaks the heap.

Step 8 / 19

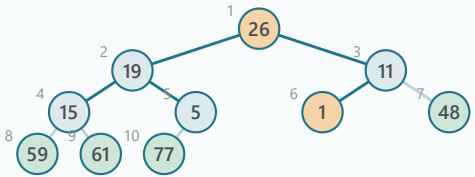
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

VARIABLES

`i = 6`

`a = [1, 19, 26, 15, 5, 11, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
26	19	11	15	5	1	48	59	61	77

heap region a[1..6] · sorted part green

heapify(a, 1, 6): 1 sank to a[6]; the new root **26** is the largest remaining.

Step 9 / 19

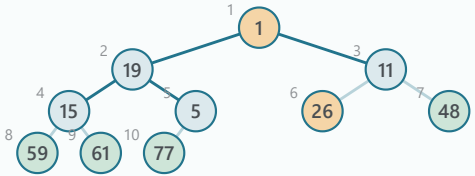
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

VARIABLES

`i = 6`

`a = [26, 19, 11, 15, 5, 1, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	19	11	15	5	26	48	59	61	77

heap region a[1..5] · sorted part green

i = 5: SWAP(a[1], a[6]) → the largest, 26, goes to the end; 1 at the root breaks the heap.

Step 10 / 19

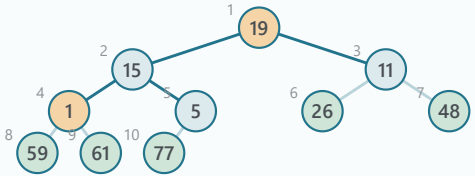
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

VARIABLES

`i = 5`

`a = [1, 19, 11, 15, 5, 26, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
19	15	11	1	5	26	48	59	61	77

heap region a[1..5] · sorted part green

heapify(a, 1, 5): 1 sank to a[4]; the new root **19** is the largest remaining.

Step 11 / 19

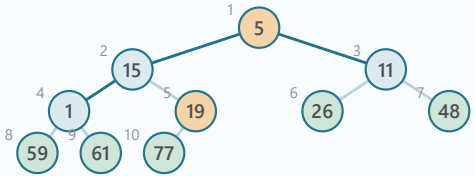
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

VARIABLES

`i = 5`

`a = [19, 15, 11, 1, 5, 26, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
5	15	11	1	19	26	48	59	61	77

heap region a[1..4] · sorted part green

i = 4: SWAP(a[1], a[5]) → the largest, 19, goes to the end; 5 at the root breaks the heap.

Step 12 / 19

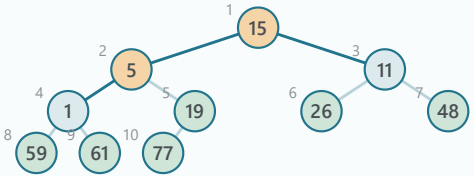
```
 1 void heapify(int a[], int root, int n) {
 2     int t = a[root], j;
 3     for (j = 2 * root; j <= n; j *= 2) {
 4         if (j < n && a[j] < a[j + 1]) j++; // larger
 5         if (t >= a[j]) break;   // t fits here
 6         a[j / 2] = a[j];        // move child up
 7     }
 8     a[j / 2] = t;
 9 }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
▶           SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

VARIABLES

`i = 4`

`a = [5, 15, 11, 1, 19, 26, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
15	5	11	1	19	26	48	59	61	77

heap region a[1..4] · sorted part green

heapify(a, 1, 4): 5 sank to a[2]; the new root **15** is the largest remaining.

Step 13 / 19

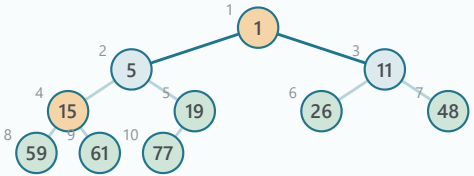
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

VARIABLES

`i = 4`

`a = [15, 5, 11, 1, 19, 26, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	5	11	15	19	26	48	59	61	77

heap region a[1..3] · sorted part green

i = 3: SWAP(a[1], a[4]) → the largest, 15, goes to the end; 1 at the root breaks the heap.

Step 14 / 19

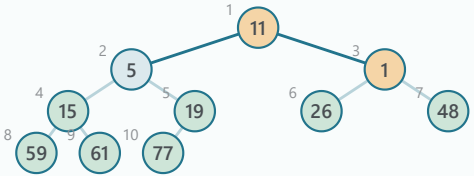
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

VARIABLES

`i = 3`

`a = [1, 5, 11, 15, 19, 26, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
11	5	1	15	19	26	48	59	61	77

heap region a[1..3] · sorted part green

heapify(a, 1, 3): 1 sank to a[3]; the new root **11** is the largest remaining.

Step 15 / 19

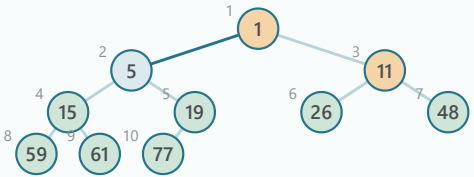
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

VARIABLES

`i = 3`

`a = [11, 5, 1, 15, 19, 26, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	5	11	15	19	26	48	59	61	77

heap region a[1..2] · sorted part green

i = 2: SWAP(a[1], a[3]) → the largest, 11, goes to the end; 1 at the root breaks the heap.

Step 16 / 19

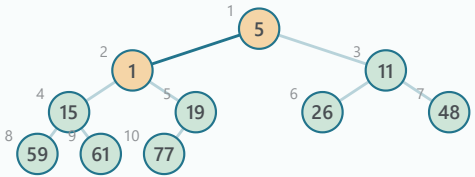
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

VARIABLES

`i = 2`

`a = [1, 5, 11, 15, 19, 26, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
5	1	11	15	19	26	48	59	61	77

heap region a[1..2] · sorted part green

heapify(a, 1, 2): 1 sank to a[2]; the new root **5** is the largest remaining.

Step 17 / 19

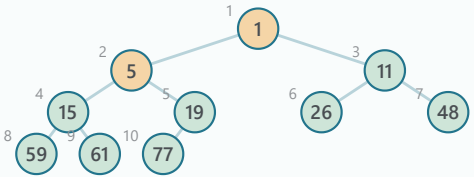
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

VARIABLES

`i = 2`

`a = [5, 1, 11, 15, 19, 26, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	5	11	15	19	26	48	59	61	77

heap region a[1..1] · sorted part green

i = 1: SWAP(a[1], a[2]) → the largest, 5, goes to the end; the single element left at the root (1) is already a heap.

Step 18 / 19

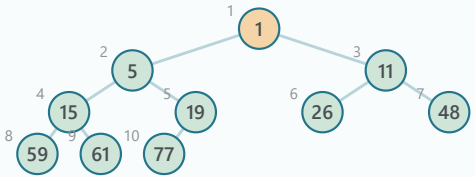
```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
▶          SWAP(a[1], a[i + 1]);
12         heapify(a, 1, i);
13     }
```

VARIABLES

`i = 1`

`a = [1, 5, 11, 15, 19, 26, 48, 59, 61, 77]`

Code Trace: The Sorting Phase, SWAP + `heapify(a, 1, i)`



1	2	3	4	5	6	7	8	9	10
1	5	11	15	19	26	48	59	61	77

heap region a[1..1] · sorted part green

heapify(a, 1, 1): j = 2 > 1, the loop never runs; 1 stays put. The array is sorted: **[1, 5, 11, 15, 19, 26, 48, 59, 61, 77]**.

Step 19 / 19

```
1  void heapify(int a[], int root, int n) {
2      int t = a[root], j;
3      for (j = 2 * root; j <= n; j *= 2) {
4          if (j < n && a[j] < a[j + 1]) j++; // larger
5          if (t >= a[j]) break;   // t fits here
6          a[j / 2] = a[j];        // move child up
7      }
8      a[j / 2] = t;
9  }
10     for (i = n - 1; i >= 1; i--) {  /* sort */
11         SWAP(a[1], a[i + 1]);
▶          heapify(a, 1, i);
13     }
```

VARIABLES

`i = 1`

`a = [1, 5, 11, 15, 19, 26, 48, 59, 61, 77]`

Heap Sort: Time Complexity

- **Rough analysis:** heapify makes at most lg n moves and is called n/2 times → building the heap costs n lg n.
- **Careful analysis:** k = number of levels ≈ lg(n+1); level i has at most 2^{i-1} nodes, each moving at most (k − i) levels:

$$\Sigma_{i=1}^{k} 2^{i-1}(k-i) = 2^k - k - 1 \leq 2n = O(n)$$

- The term $\Sigma\, i \cdot 2^{i-1}$ comes from differentiating the geometric series $S(y) = (y^{k+1} - 1)/(y - 1)$ and setting $y = 2$.
- The sorting phase: n − 1 calls of heapify(a, 1, i) → O(n lg n). Total **O(n lg n)**, also in the worst case; O(1) extra memory; not stable.
- A bound of "≤ 2n" means **O(n)**, not o(n) (being bounded by 2n is not little-o).

k	$\Sigma\, 2^{i-1}(k-i)$	2^k-k-1	$n = 2^k-1$
2	1	1	3
3	4	4	7
4	11	11	15
6	57	57	63
10	1013	1013	1023

INSTRUCTOR "When we implemented Kruskal, we built the heap by inserting the elements into a min heap one by one. That does produce a heap. But here we turned the array holding the elements into a min heap at about half that cost."

Heap Sort Variants, and Knowing Your Data

When?

For fewer than 50,000 items heap sort is a good choice (≈ 2n lg n comparisons). For larger data there is **bottom-up heapsort**, which makes fewer comparisons, and theoretically even better variants.

The bottom-up idea

The hole at the root sinks along the larger children to a leaf without comparing with t; then t is placed by moving up. Mostly one comparison per level instead of two.

INSTRUCTOR "As with counting sort, if we know some properties of the data to sort, we can choose an approach accordingly. In that sense Shell sort and bucket sort are used together. You should know the basic few."

Radix Sort

- Sorting by several keys: e.g. people by **surname**, and those with the same surname by **first name**.
- Keys K_0, K_1, …, K_{r-1}: K_0 is the **most significant** (MSD), K_{r-1} the **least significant** (LSD). In the example, the surname is most significant.
- **LSD radix sort:** sort first by the least significant digit, then by more significant ones in turn, and finally by the MSD.
- Each pass must be **stable**: the order from the previous pass is kept for equal digits.
- It never compares numbers with each other → a *non-comparison* sort.

```
#define MAX_DIGIT 3
#define RADIX 10
typedef struct node *node_ptr;
struct node {
    int digit[MAX_DIGIT];  /* 179 → {1, 7, 9} */
    node_ptr next;
};
```

K_0: hundreds (MSD) · K_1: tens · K_2: units (LSD). Range [0, 999]. A typedef with no type name, like `typedef struct node {…};`, is meaningless.

Code Trace: `radixsort`, i = 2 (Units)



Pass i = 2 (units digit): the 10 buckets are emptied.

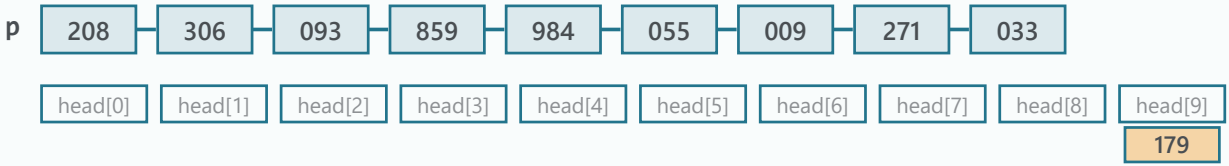
Step 1 / 12

```
▶ for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2     for (int j = 0; j < RADIX; j++)
▶         head[j] = tail[j] = NULL;
4     for (; p; p = p->next) {    // scatter
5         int b = p->digit[i];
6         if (!head[b]) head[b] = p;
7         else tail[b]->next = p;
8         tail[b] = p;
9     }
10    for (int j = RADIX - 1; j >= 0; j--) // gather
11        if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `p = 179→208→306→093…`

Code Trace: `radixsort`, i = 2 (Units)



179: digit[2] = 9 → appended to the **tail** of bucket 9 (arrival order is kept → stable).

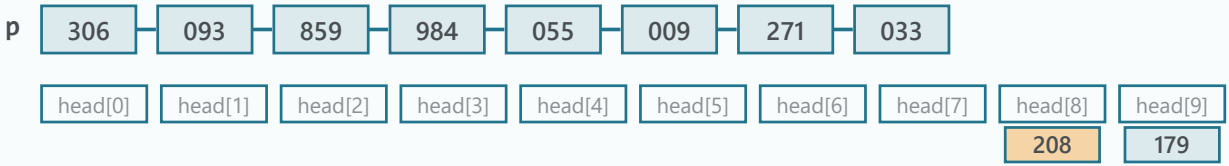
Step 2 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
▶          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `b = 9` `p = 179`

Code Trace: `radixsort`, i = 2 (Units)



208: digit[2] = 8 → appended to the **tail** of bucket 8 (arrival order is kept → stable).

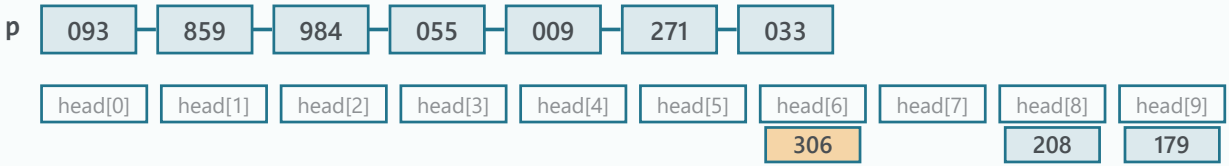
Step 3 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
▶          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `b = 8` `p = 208`

Code Trace: `radixsort`, i = 2 (Units)



306: digit[2] = 6 → appended to the **tail** of bucket 6 (arrival order is kept → stable).

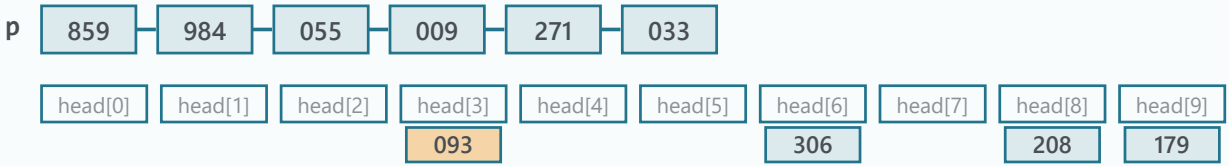
Step 4 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
▶          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `b = 6` `p = 306`

Code Trace: `radixsort`, i = 2 (Units)



093: digit[2] = 3 → appended to the **tail** of bucket 3 (arrival order is kept → stable).

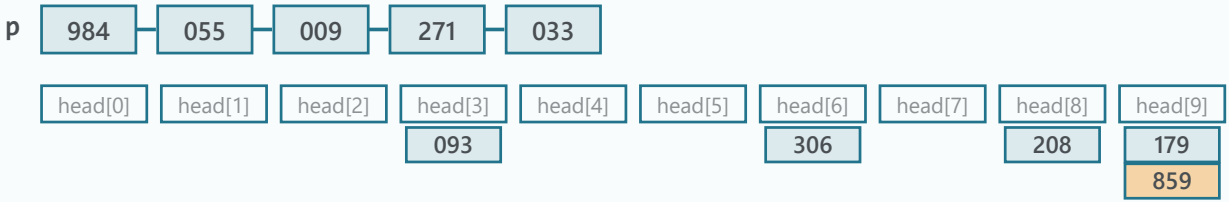
Step 5 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
▶          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `b = 3` `p = 093`

Code Trace: `radixsort`, i = 2 (Units)



859: digit[2] = 9 → appended to the **tail** of bucket 9 (arrival order is kept → stable).

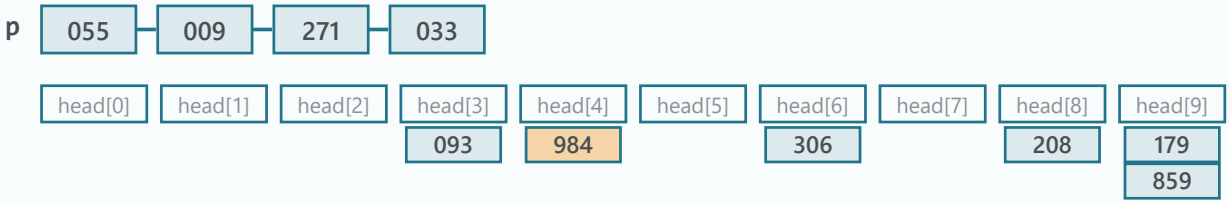
Step 6 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
▶          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `b = 9` `p = 859`

Code Trace: `radixsort`, i = 2 (Units)



984: digit[2] = 4 → appended to the **tail** of bucket 4 (arrival order is kept → stable).

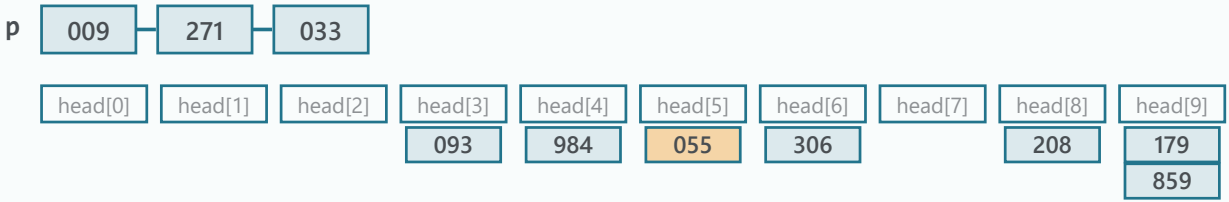
Step 7 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
▶          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `b = 4` `p = 984`

Code Trace: `radixsort`, i = 2 (Units)



055: digit[2] = 5 → appended to the **tail** of bucket 5 (arrival order is kept → stable).

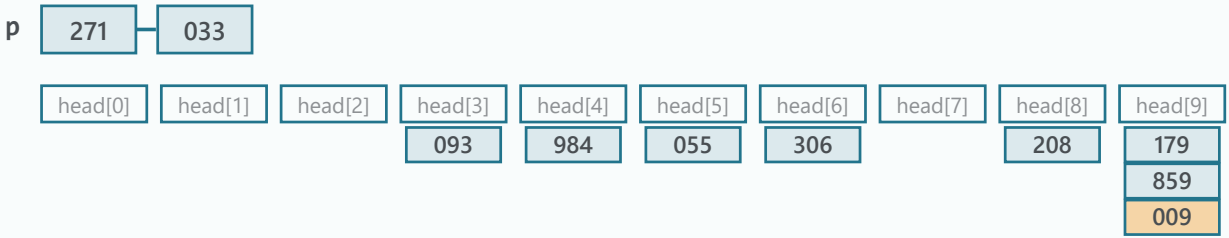
Step 8 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
▶          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `b = 5` `p = 055`

Code Trace: `radixsort`, i = 2 (Units)



009: digit[2] = 9 → appended to the **tail** of bucket 9 (arrival order is kept → stable).

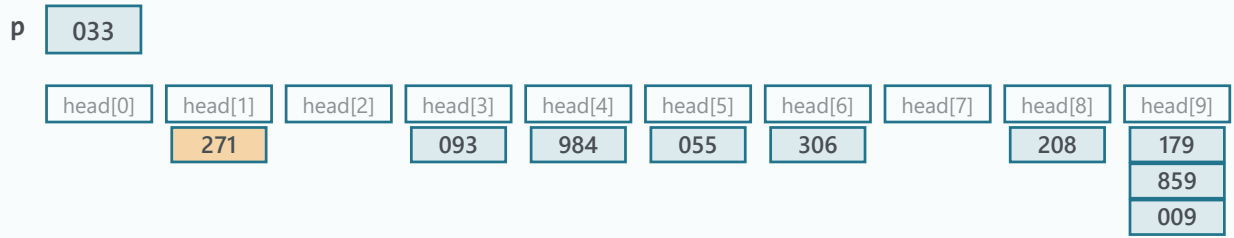
Step 9 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
▶          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `b = 9` `p = 009`

Code Trace: `radixsort`, i = 2 (Units)



271: digit[2] = 1 → appended to the **tail** of bucket 1 (arrival order is kept → stable).

Step 10 / 12

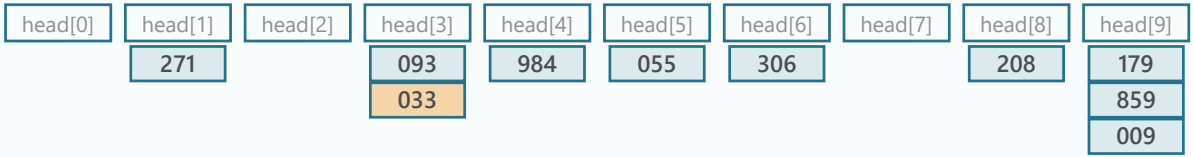
```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
▶          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `b = 1` `p = 271`

Code Trace: `radixsort`, i = 2 (Units)

p NULL



033: digit[2] = 3 → appended to the **tail** of bucket 3 (arrival order is kept → stable).

Step 11 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
▶          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `b = 3` `p = 033`

Code Trace: `radixsort`, i = 2 (Units)



Gather: buckets are visited from 9 down to 0 and each is linked to the **front** of the list → the smallest bucket ends up first: 271 093 033 984 055 306 208 179 859 009.

Step 12 / 12

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
5          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
8          tail[b] = p;
9      }
▶      for (int j = RADIX - 1; j >= 0; j--) // gather
▶          if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 2` `p = 271`

Code Trace (Continued): i = 1 and i = 0



Pass i = 1 (tens digit): the 10 buckets are emptied.

Step 1 / 7

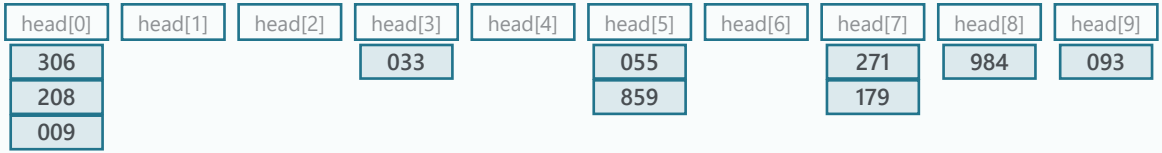
```
▶ for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2     for (int j = 0; j < RADIX; j++)
▶         head[j] = tail[j] = NULL;
4     for (; p; p = p->next) {    // scatter
5         int b = p->digit[i];
6         if (!head[b]) head[b] = p;
7         else tail[b]->next = p;
8         tail[b] = p;
9     }
10    for (int j = RADIX - 1; j >= 0; j--) // gather
11        if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 1` `p = 271→093→033→984…`

Code Trace (Continued): i = 1 and i = 0

p NULL



Scatter: each item goes to the tail of the bucket for its tens digit.

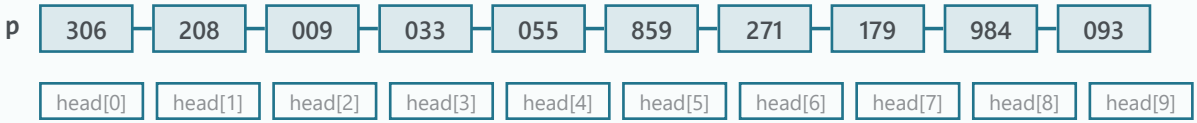
Step 2 / 7

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
▶      for (; p; p = p->next) {    // scatter
5          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

i = 1

Code Trace (Continued): i = 1 and i = 0



Gather: buckets are visited from 9 down to 0 and each is linked to the **front** of the list → the smallest bucket ends up first: 306 208 009 033 055 859 271 179 984 093.

Step 3 / 7

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
5          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
8          tail[b] = p;
9      }
▶      for (int j = RADIX - 1; j >= 0; j--) // gather
▶          if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

`i = 1` `p = 306`

Code Trace (Continued): i = 1 and i = 0



Pass i = 0 (hundreds digit): the 10 buckets are emptied.

Step 4 / 7

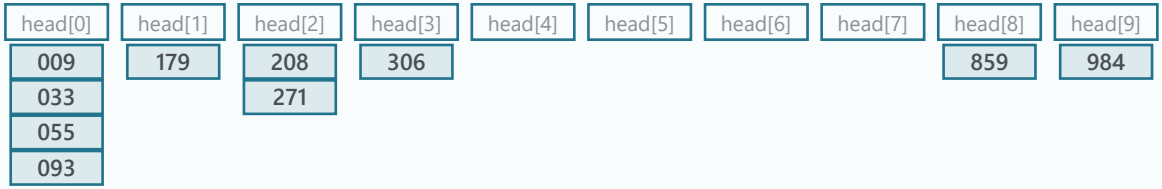
```
▶ for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2     for (int j = 0; j < RADIX; j++)
▶         head[j] = tail[j] = NULL;
4     for (; p; p = p->next) {    // scatter
5         int b = p->digit[i];
6         if (!head[b]) head[b] = p;
7         else tail[b]->next = p;
8         tail[b] = p;
9     }
10    for (int j = RADIX - 1; j >= 0; j--) // gather
11        if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

i = 0 | p = 306→208→009→033…

Code Trace (Continued): i = 1 and i = 0

p NULL



Scatter: each item goes to the tail of the bucket for its hundreds digit.

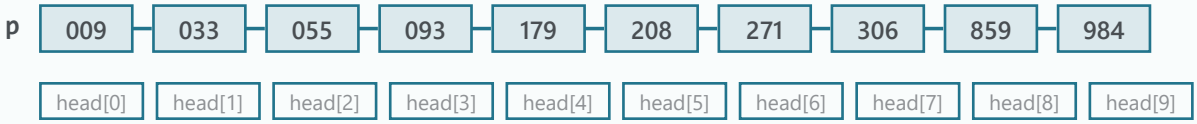
Step 5 / 7

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
▶      for (; p; p = p->next) {    // scatter
5          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
▶          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

i = 0

Code Trace (Continued): i = 1 and i = 0



Gather: buckets are visited from 9 down to 0 and each is linked to the **front** of the list → the smallest bucket ends up first: 009 033 055 093 179 208 271 306 859 984.

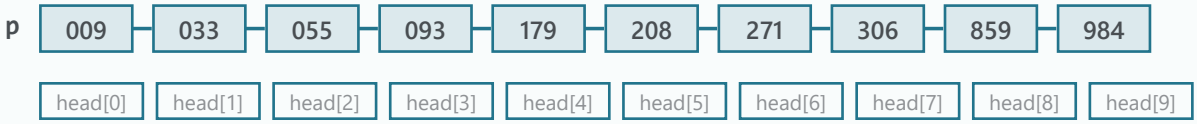
Step 6 / 7

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
5          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
8          tail[b] = p;
9      }
▶      for (int j = RADIX - 1; j >= 0; j--) // gather
▶          if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
13 return p;
```

VARIABLES

i = 0 p = 009

Code Trace (Continued): i = 1 and i = 0



After three passes the list is sorted; the head pointer is returned.

Step 7 / 7

```
1  for (int i = MAX_DIGIT - 1; i >= 0; i--) {
2      for (int j = 0; j < RADIX; j++)
3          head[j] = tail[j] = NULL;
4      for (; p; p = p->next) {    // scatter
5          int b = p->digit[i];
6          if (!head[b]) head[b] = p;
7          else tail[b]->next = p;
8          tail[b] = p;
9      }
10     for (int j = RADIX - 1; j >= 0; j--) // gather
11         if (head[j]) { tail[j]->next = p; p = head[j]; }
12 }
▶  return p;
```

VARIABLES

p = 009

Radix Sort: Analysis

$$O(MAX_DIGIT \cdot (RADIX + n + RADIX)) = O(MAX_DIGIT \cdot (RADIX + n))$$

- In the example O(3 · (10 + n)): it looks linear, but like counting sort it is a **special case**: the number of digits and the radix are taken as constants.
- The data must be (a) nonnegative, (b) integers, (c) in a small range. With a larger range it still sorts, but the cost grows fast.
- Extra memory: 2·RADIX pointers (the list is relinked in place).

INSTRUCTOR "Sorting only integers is a drawback. It can be adapted to floating-point types, but by the time you finish adapting it, the other algorithms will have finished sorting."

pass	list
initial	179 208 306 093 859 984 055 009 271 033
i = 2	271 093 033 984 055 306 208 179 859 009
i = 1	306 208 009 033 055 859 271 179 984 093
i = 0	009 033 055 093 179 208 271 306 859 984

Exam Question: a Modified Radix Sort

A student, thinking the function should handle the most significant digit first, changed the outer loop to `for (i = 0; i ≤ MAX_DIGIT−1; i++)` and the gather loop to `for (j = 0; j ≤ RADIX−1; j++)`. Run the function on the list below, write the list after each step and interpret the result.

```
p → 346 180 008 930 023 451 670 098
    203 007
```

▶ **Solution**

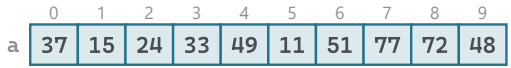
Lab: Advanced Sorts

Quick sort: 37, 15, 24, 33, 49, 11, 51, 77, 72, 48

Input (comma-separated numbers) `37, 15, 24, 33, 49, 11, 51, 77, 72, 48` Apply

CODE: quicksort1

```
1 void quicksort1(int a[], int L, int R) {
2     if (R <= L) return;
3     int i = partition(a, L, R);
4     quicksort1(a, L, i - 1);
5     quicksort1(a, i + 1, R);
6 }
```



`comparisons 0`

`OUTPUT` (none yet)

Input: [37, 15, 24, 33, 49, 11, 51, 77, 72, 48]. Quick sort: `partition` puts the pivot (a[R]) in its final place, with smaller items to its left and larger ones to its right. Both sides are sorted recursively.

SUMMARY

Complexity Summary and Comparison

algorithm	best	average	worst	extra memory	stable	in place
Selection	n^2	n^2	n^2	O(1)	no	yes
Insertion	n	n^2	n^2	O(1)	yes	yes
Bubble (early exit)	n	n^2	n^2	O(1)	yes	yes
Shell (Knuth)	n lg n	—	$n^{3/2}$	O(1)	no	yes
Quicksort	n lg n	1.39 n lg n	n^2	O(lg n) stack*	no	yes
Merge sort	n lg n	n lg n	n lg n	O(n)	yes**	no
Heap sort	n lg n	n lg n	n lg n	O(1)	no	yes
Counting sort	n + k	n + k	n + k	O(n + k)	yes	no
Radix (d digits)	d(n + r)	d(n + r)	d(n + r)	O(r)***	yes	—

* If the smaller piece is processed first; otherwise O(n) in the worst case. ** With the classic merge. *** On a linked list. The lower bound for comparison sorting is $\lceil \lg n! \rceil \approx n \lg n - 1.44\, n$.

SUMMARY

Which One When?

General purpose, in memory

Quicksort (median-of-3 + insertion sort on small pieces) or the library `qsort`.

Worst-case guarantee

Heap sort (in place) or **introsort**.

Stability / linked lists / disk

Merge sort: stable, merges sorted runs, the basis of external sorting.

Large records

Sort **indices/links**, not the records (linked merge sort).

Fixed-length integers/keys

Radix: d passes × O(n + r).

Small range

Counting sort.

SUMMARY

Common Mistakes

1. Forgetting the `j == L` test in the j loop of partition(): if the pivot is the smallest, the scan runs off the start of the array.

2. Always taking a[R] as the pivot on sorted input: the depth becomes n and the stack overflows. Cure: median-of-3 or a random pivot.

3. Pushing the pieces in arbitrary order in the non-recursive version: the stack grows to O(n).

4. Calling `malloc` for `b[]` on every merge sort call (slow), or never allocating it.

5. Mixing 1-based heap indices with a 0-based array in heapify: children 2i, 2i+1 are right only in a[1..n].

6. `return x - y;` in a qsort comparator: overflow; or an inconsistent (non-transitive) comparator.

SUMMARY

Exercises

1. Write the array and the return value of `partition(a, 0, 4)` for [5, 9, 1, 7, 3].
2. How many partition() calls does quicksort make on a sorted array with n = 8? What is the recursion depth?
3. How many merge calls does merge sort make for 16 elements? How many comparisons at most?
4. Write the array after building the heap for [4, 10, 3, 5, 1] (a[1..5]).
5. Write the order of 170, 45, 75, 90, 802, 24, 2, 66 after the units pass of radix sort.
6. Why is heap sort not stable? Give a small example.

▶ **Show answers**

SUMMARY

Chapter Summary

Quicksort

Places the pivot and splits the problem: best n lg n, average 1.39 n lg n, worst n^2.

Improvements

Smaller piece first (lg n stack), median-of-3, insertion sort on small pieces; qsort and introsort.

Merge sort

n lg n in every case, O(n) extra memory; the linked version never copies large records.

Heap sort

Building the heap is O(n), sorting O(n lg n); in place but not stable.

Radix sort

Stable buckets from LSD to MSD: O(d(n + r)); nonnegative integers in a small range.

Analysis tools

Recurrences, iteration, the subtraction trick, differentiating a series: all the course's analysis tools meet here.

SUMMARY

References

- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapters 7–10 (quicksort, mergesort, priority queues/heapsort, radix sorting).
- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Chapter 7.
- T. H. Cormen et al., *Introduction to Algorithms*, Chapters 6–8; D. Musser, "Introspective Sorting and Selection Algorithms", 1997.
- ISO/IEC 9899 (the C standard), definition of `qsort` (§7.22.5.2).

← Chapter 19: Basic Sorting Algorithms

Course page

The End

Chapter 20: Advanced Sorting Algorithms

"If this orphaned work stays with the master, / shame on the apprentice who does not carry it on!" — *N. F. K.*

End of the course · **Course page**