Chapter 19: Basic Sorting Algorithms

Data Structures and Algorithms with C

Stability · Selection · Insertion · Average case · Bubble · Counting · Shell · Bucket · Lower bound

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_19_Basic_Sorting_Algorithms.html

Learning Outcomes

Concepts

Distinguish records, keys, internal/external sorting, **stability** and **in-place** sorting.

Three basic algorithms

Write selection, insertion and bubble sort in C and trace them step by step.

Analysis

Count comparisons and exchanges; compute the best, worst and **average case**.

Non-comparison sorting

Explain when counting sort and bucket sort run in O(n).

Shell sort

Show why insertion sort with gaps beats plain insertion sort.

Lower bound

Prove the Ω(n lg n) bound for comparison sorting with a decision tree.

Sorting: Records, Keys, Internal/External

- **Records** made of *fields* form a **list** when kept in memory and a **file** when kept outside it.
- The field(s) that identify a record **uniquely** form the **key**; sorting is done by key.
- **Internal sorting**: all the data is in memory. **External sorting**: auxiliary storage such as disk or tape is used; the number of disk accesses dominates.

record = fields

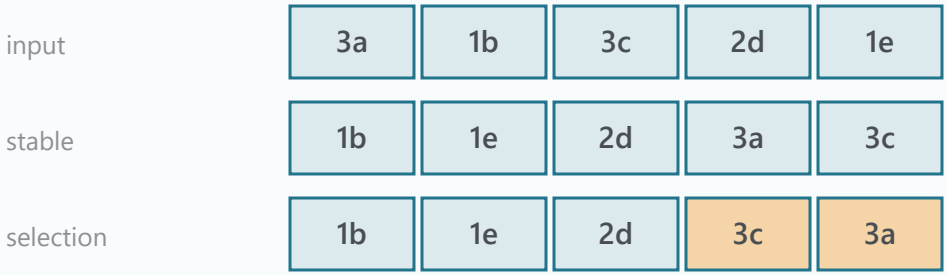
no. (key)	name	surname	avg.
1042	Ayşe	Yılmaz	3.1
0977	Can	Demir	2.8
1310	Ece	Kaya	3.6

in memory: list · outside memory: file

Key: student number

INSTRUCTOR "We will deal with internal sorting. Along the way we will also lay a foundation for external sorting."

Stability and In-Place Sorting



3a was ahead of 3c; after selection sort 3c is ahead

Stable

Records with equal keys keep their **relative order**. Sort stably by first name, then by surname, and people with the same surname stay in first-name order (the basis of radix sort).

In-place

Uses only **O(1)** (by some definitions O(lg n)) extra memory beyond the input. Selection, insertion and bubble sort are in place; counting sort is not (arrays c and b).

Common Setup: `less` and `SWAP`

```
#define N 15
#define less(A, B) ((A) < (B))
#define SWAP(A, B) do { int t_ = (A); (A) = (B); (B) = t_; } while (0)
```

```
int main(void) {
    int *a = malloc(N * sizeof *a);
    if (a == NULL) return 1;
    for (int i = 0; i < N; i++)
        a[i] = rand() % 50;
    selection(a, 0, N - 1);   /* call the sort */
    for (int i = 0; i < N; i++) printf("%d ", a[i]);
    free(a);
    return 0;
}
```

A macro trap

Writing the macro as `{int t = A; A=B; B=t;}` fails to compile inside `if (…) SWAP(…); else` ; `do { … } while (0)` closes that trap. The idiom `if (less(A, B)) SWAP(A, B);` swaps **when A < B** (compare-exchange).

Selection Sort

```
void selection(int a[], int L, int R) {
    for (int i = L; i < R; i++) {
        int min = i;
        for (int j = i + 1; j <= R; j++)
            if (less(a[j], a[min]))
                min = j;
        SWAP(a[i], a[min]);
    }
}
```

1. Find the smallest element in the list.
2. Swap it with the element in the first position.
3. Repeat on the remaining sublist, starting from the 2nd element.

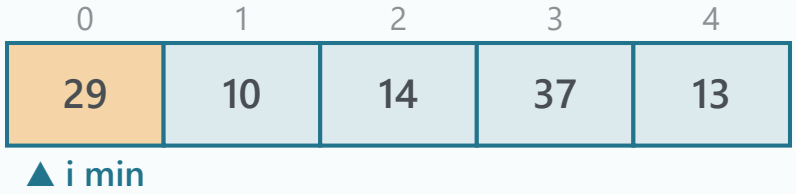
Invariant

After each outer pass, `a[L..i]` is sorted and holds the smallest elements of the array.

Why scan to the end?

To pick the minimum, the remaining sublist **must** be scanned to its end, even if the data is already sorted.

Code Trace: `selection(a, 0, 4)`



Pass i = 0: the remaining sublist is a[0..4]. Initially the minimum's index is **min = 0** (a[0] = 29).

Step 1 / 18

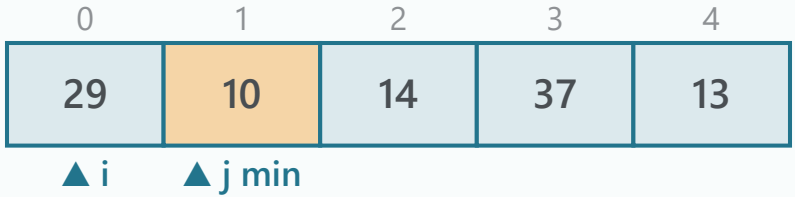
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
▶         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (less(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 0` `min = 0`

`a = [29, 10, 14, 37, 13]`

Code Trace: `selection(a, 0, 4)`



a[1] = 10 < a[0] = 29 → new minimum: **min = 1**.

Step 2 / 18

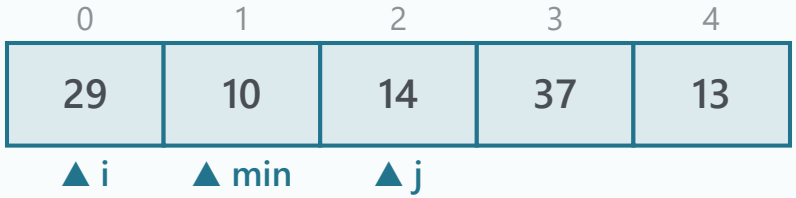
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶               if (less(a[j], a[min]))
▶                   min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 0` `j = 1` `min = 1`

`a = [29, 10, 14, 37, 13]`

Code Trace: `selection(a, 0, 4)`



a[2] = 14 ≥ a[1] = 10 → min unchanged.

Step 3 / 18

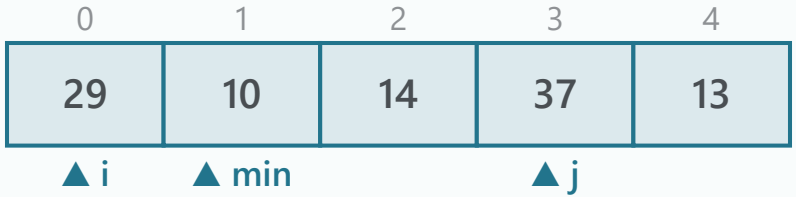
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (less(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 0` `j = 2` `min = 1`

`a = [29, 10, 14, 37, 13]`

Code Trace: `selection(a, 0, 4)`



a[3] = 37 ≥ a[1] = 10 → min unchanged.

Step 4 / 18

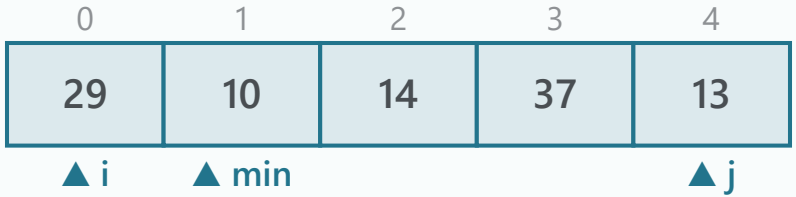
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (less(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 0` `j = 3` `min = 1`

`a = [29, 10, 14, 37, 13]`

Code Trace: `selection(a, 0, 4)`



a[4] = 13 ≥ a[1] = 10 → min unchanged.

Step 5 / 18

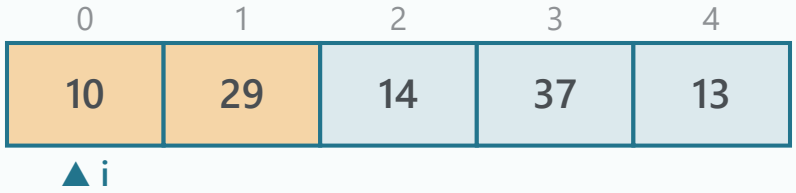
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (less(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 0` `j = 4` `min = 1`

`a = [29, 10, 14, 37, 13]`

Code Trace: `selection(a, 0, 4)`



SWAP(a[0], a[1]): 10 is in place; a[0..0] is now sorted.

Step 6 / 18

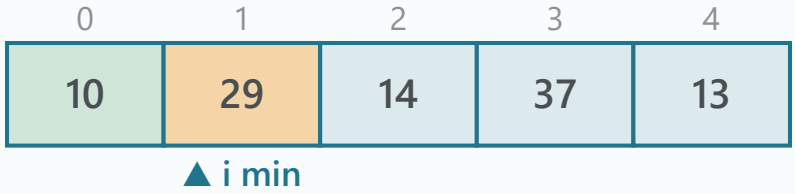
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (less(a[j], a[min]))
6                 min = j;
▶           SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 0` `min = 1`

`a = [10, 29, 14, 37, 13]`

Code Trace: `selection(a, 0, 4)`



Pass i = 1: the remaining sublist is a[1..4]. Initially the minimum's index is **min = 1** (a[1] = 29).

Step 7 / 18

```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
▶         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (less(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 1` `min = 1`

`a = [10, 29, 14, 37, 13]`

Code Trace: `selection(a, 0, 4)`



a[2] = 14 < a[1] = 29 → new minimum: **min = 2**.

Step 8 / 18

```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶               if (less(a[j], a[min]))
▶                   min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 1` `j = 2` `min = 2`

`a = [10, 29, 14, 37, 13]`

Code Trace: `selection(a, 0, 4)`



a[3] = 37 ≥ a[2] = 14 → min unchanged.

Step 9 / 18

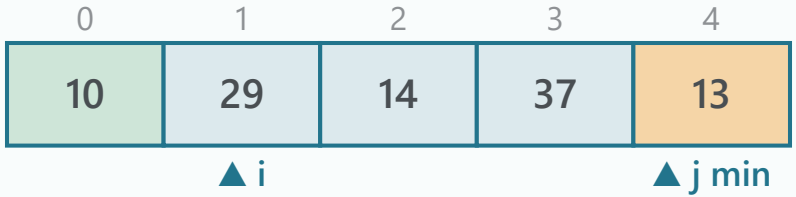
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (less(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 1` `j = 3` `min = 2`

`a = [10, 29, 14, 37, 13]`

Code Trace: `selection(a, 0, 4)`



a[4] = 13 < a[2] = 14 → new minimum: **min = 4**.

Step 10 / 18

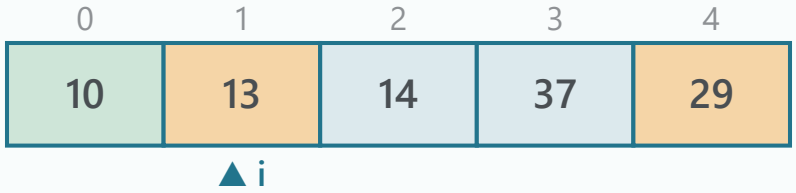
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶               if (less(a[j], a[min]))
▶                   min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 1` `j = 4` `min = 4`

`a = [10, 29, 14, 37, 13]`

Code Trace: `selection(a, 0, 4)`



SWAP(a[1], a[4]): 13 is in place; a[0..1] is now sorted.

Step 11 / 18

```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (less(a[j], a[min]))
6                 min = j;
▶           SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 1` `min = 4`

`a = [10, 13, 14, 37, 29]`

Code Trace: `selection(a, 0, 4)`



Pass i = 2: the remaining sublist is a[2..4]. Initially the minimum's index is **min = 2** (a[2] = 14).

Step 12 / 18

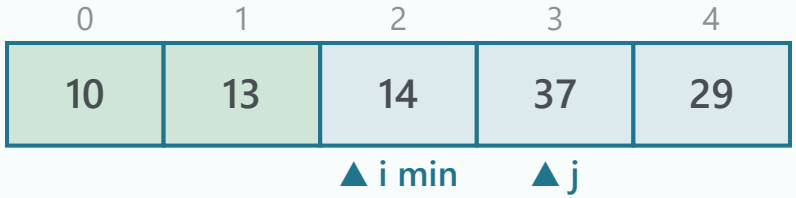
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (less(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 2` `min = 2`

`a = [10, 13, 14, 37, 29]`

Code Trace: `selection(a, 0, 4)`



a[3] = 37 ≥ a[2] = 14 → min unchanged.

Step 13 / 18

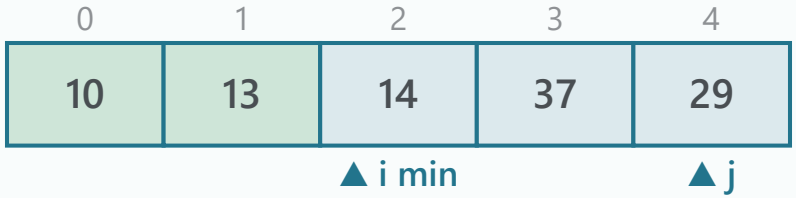
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (less(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 2` `j = 3` `min = 2`

`a = [10, 13, 14, 37, 29]`

Code Trace: `selection(a, 0, 4)`



a[4] = 29 ≥ a[2] = 14 → min unchanged.

Step 14 / 18

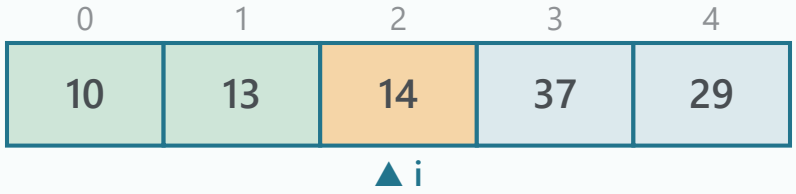
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶             if (less(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 2` `j = 4` `min = 2`

`a = [10, 13, 14, 37, 29]`

Code Trace: `selection(a, 0, 4)`



min = i, so the element is swapped with itself (no change); a[0..2] is sorted.

Step 15 / 18

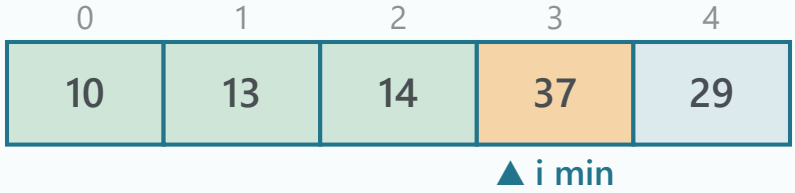
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (less(a[j], a[min]))
6                 min = j;
          SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 2` `min = 2`

`a = [10, 13, 14, 37, 29]`

Code Trace: `selection(a, 0, 4)`



Pass i = 3: the remaining sublist is a[3..4]. Initially the minimum's index is **min = 3** (a[3] = 37).

Step 16 / 18

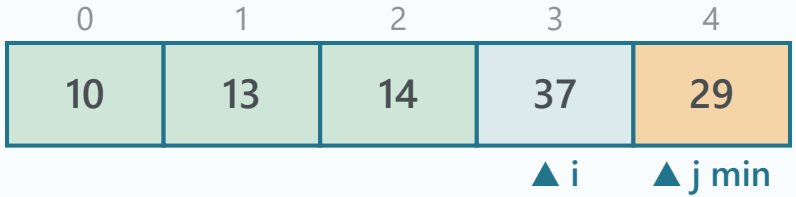
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
▶           int min = i;
4           for (int j = i + 1; j <= R; j++)
5               if (less(a[j], a[min]))
6                   min = j;
7           SWAP(a[i], a[min]);
8       }
9   }
```

VARIABLES

`i = 3` `min = 3`

`a = [10, 13, 14, 37, 29]`

Code Trace: `selection(a, 0, 4)`



a[4] = 29 < a[3] = 37 → new minimum: **min = 4**.

Step 17 / 18

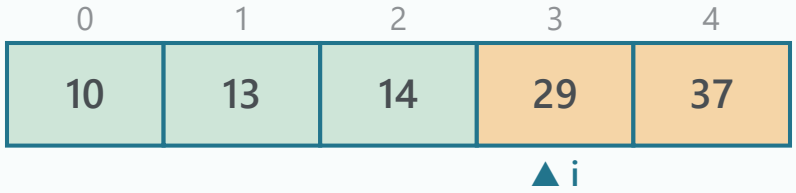
```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
▶               if (less(a[j], a[min]))
▶                   min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 3` `j = 4` `min = 4`

`a = [10, 13, 14, 37, 29]`

Code Trace: `selection(a, 0, 4)`



SWAP(a[3], a[4]): 29 is in place; a[0..3] is now sorted.

Step 18 / 18

```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (less(a[j], a[min]))
6                 min = j;
▶           SWAP(a[i], a[min]);
8     }
9 }
```

VARIABLES

`i = 3` `min = 4`

`a = [10, 13, 14, 29, 37]`

Analysis of Selection Sort

$$(n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2 \approx n^2/2$$

- The number of comparisons is **independent** of the input: best = average = worst ≈ $n^2/2$.
- The number of exchanges is **n − 1**: when the last element is reached, it is already in place.
- Complexity **$O(n^2)$**; O(1) extra memory (in place).
- Long-distance swaps make it **not stable** (the earlier 3a/3c example).

n	comparisons	swaps
5	10	4
10	45	9
100	4950	99
1000	499500	999

When swaps are expensive (large records, flash memory writes), the few swaps of selection sort make it valuable.

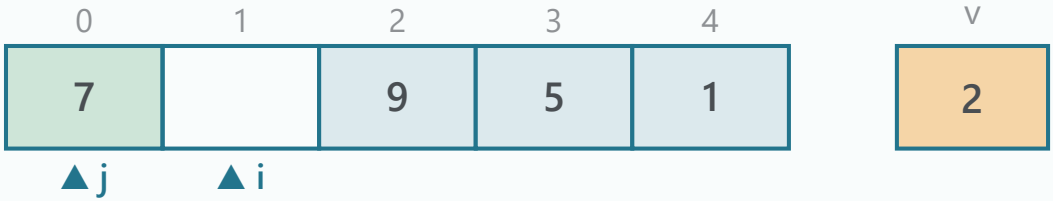
Insertion Sort: Two Versions

```
void insertion(int a[], int L, int R) {
    for (int i = L + 1; i <= R; i++) {
        int v = a[i];       // to insert
        int j = i - 1;      // a[L..i-1] sorted
        while (j >= L && a[j] > v) {
            a[j + 1] = a[j]; // shift right
            j--;
        }
        a[j + 1] = v;
    }
}
```

```
void insertion1(int a[], int L, int R) {
    for (int i = R; i > L; i--) // sentinel
        if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
    for (int i = L + 2; i <= R; i++) {
        int v = a[i], j = i;
        while (less(v, a[j - 1])) {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = v;
    }
}
```

- Each pass takes one element from the input and inserts it at the right place in the **sorted prefix** (like arranging playing cards).
- `insertion1` on the right is the **improved version** of the left one: it first moves the minimum to `a[L]`; thanks to this **sentinel**, the inner loop needs no `j > L` test.
- Writing the condition as `j ≥ 0` is wrong; if the subarray starts at `L > 0` it overruns. The correct test is `j ≥ L`.

Code Trace: `insertion(a, 0, 4)`



i = 1: **v = a[1] = 2** is set aside; a[0..0] is sorted; find a place for v.

Step 1 / 15

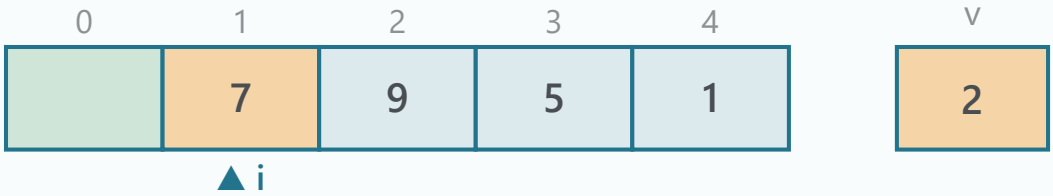
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
▶          int v = a[i];      // to insert
▶          int j = i - 1;     // a[L..i-1] sorted
5          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // shift right
7              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 1` `v = 2` `j = 0`

`a = [7, 2, 9, 5, 1]`

Code Trace: `insertion(a, 0, 4)`



a[0] = 7 > v = 2 → 7 shifts one right (a[1]); j = -1.

Step 2 / 15

```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // shift right
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 1` `v = 2` `j = -1`

`a = [7, 7, 9, 5, 1]`

Code Trace: `insertion(a, 0, 4)`



j = −1, the start of the array → the loop ends; **a[0] = 2**. The sorted prefix is now a[0..1].

Step 3 / 15

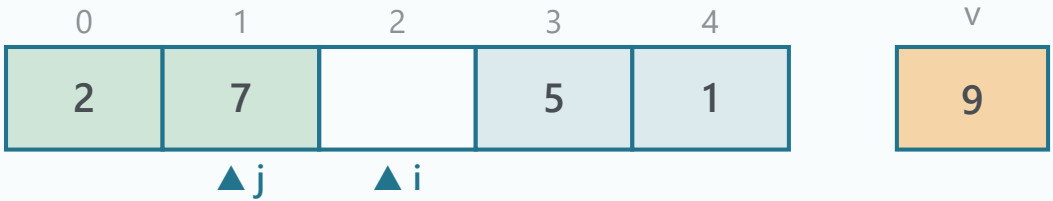
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // shift right
7              j--;
8          }
▶          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 1` `v = 2` `j = -1`

`a = [2, 7, 9, 5, 1]`

Code Trace: `insertion(a, 0, 4)`



i = 2: **v = a[2] = 9** is set aside; a[0..1] is sorted; find a place for v.

Step 4 / 15

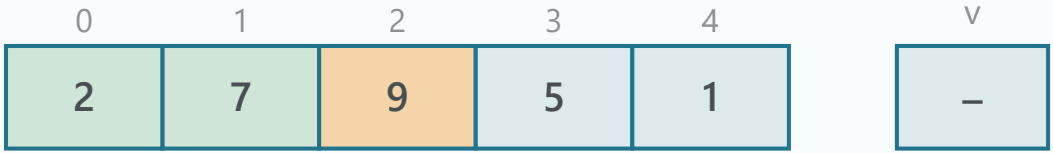
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
▶          int v = a[i];       // to insert
▶          int j = i - 1;      // a[L..i-1] sorted
5          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // shift right
7              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 2` `v = 9` `j = 1`

`a = [2, 7, 9, 5, 1]`

Code Trace: `insertion(a, 0, 4)`



a[1] = 7 ≤ v → the loop ends; **a[2] = 9**. The sorted prefix is now a[0..2].

Step 5 / 15

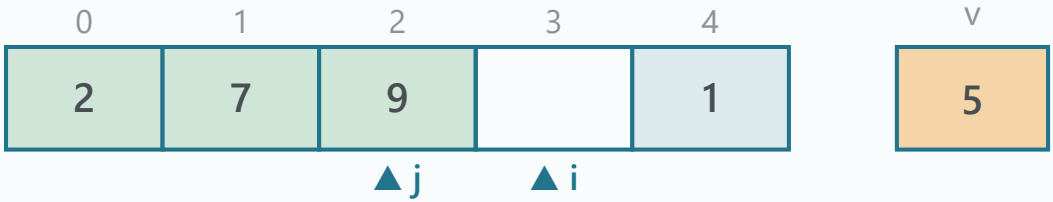
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // shift right
7              j--;
8          }
▶          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 2` `v = 9` `j = 1`

`a = [2, 7, 9, 5, 1]`

Code Trace: `insertion(a, 0, 4)`



i = 3: **v = a[3] = 5** is set aside; a[0..2] is sorted; find a place for v.

Step 6 / 15

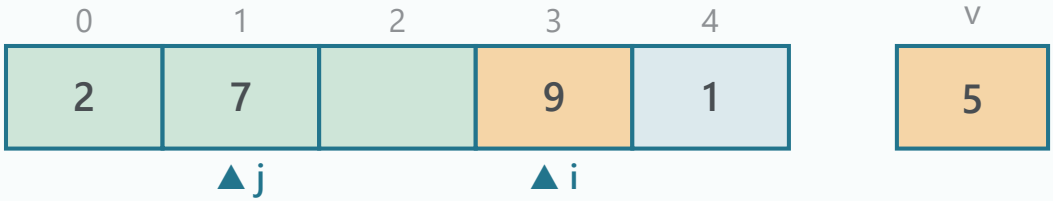
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
▶          int v = a[i];       // to insert
▶          int j = i - 1;      // a[L..i-1] sorted
5          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // shift right
7              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 3` `v = 5` `j = 2`

`a = [2, 7, 9, 5, 1]`

Code Trace: `insertion(a, 0, 4)`



a[2] = 9 > v = 5 → 9 shifts one right (a[3]); j = 1.

Step 7 / 15

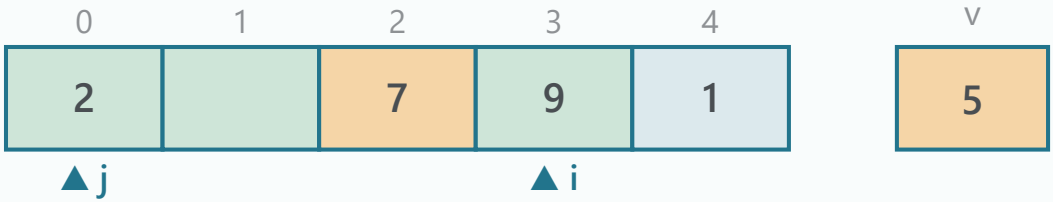
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // shift right
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 3` `v = 5` `j = 1`

`a = [2, 7, 9, 9, 1]`

Code Trace: `insertion(a, 0, 4)`



a[1] = 7 > v = 5 → 7 shifts one right (a[2]); j = 0.

Step 8 / 15

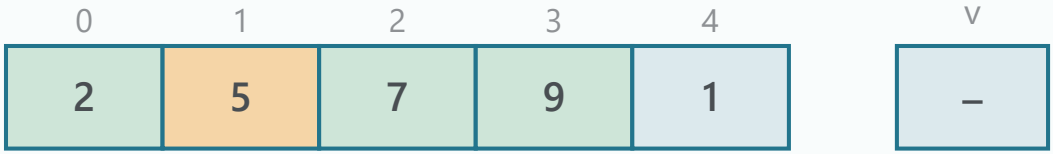
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // shift right
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 3` `v = 5` `j = 0`

`a = [2, 7, 7, 9, 1]`

Code Trace: `insertion(a, 0, 4)`



a[0] = 2 ≤ v → the loop ends; **a[1] = 5**. The sorted prefix is now a[0..3].

Step 9 / 15

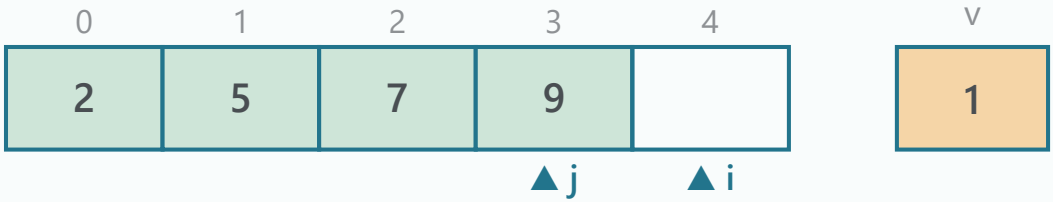
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // shift right
7              j--;
8          }
▶          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 3` `v = 5` `j = 0`

`a = [2, 5, 7, 9, 1]`

Code Trace: `insertion(a, 0, 4)`



i = 4: **v = a[4] = 1** is set aside; a[0..3] is sorted; find a place for v.

Step 10 / 15

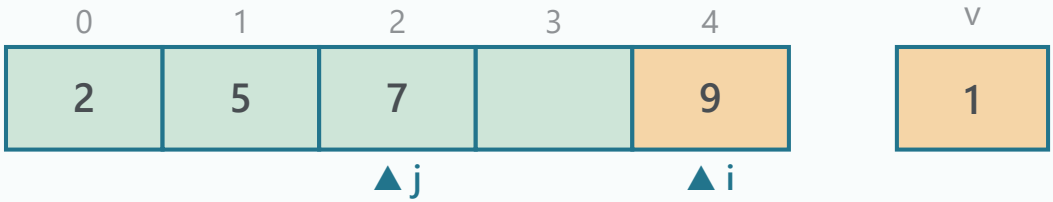
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
▶          int v = a[i];       // to insert
▶          int j = i - 1;      // a[L..i-1] sorted
5          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // shift right
7              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 4` `v = 1` `j = 3`

`a = [2, 5, 7, 9, 1]`

Code Trace: `insertion(a, 0, 4)`



a[3] = 9 > v = 1 → 9 shifts one right (a[4]); j = 2.

Step 11 / 15

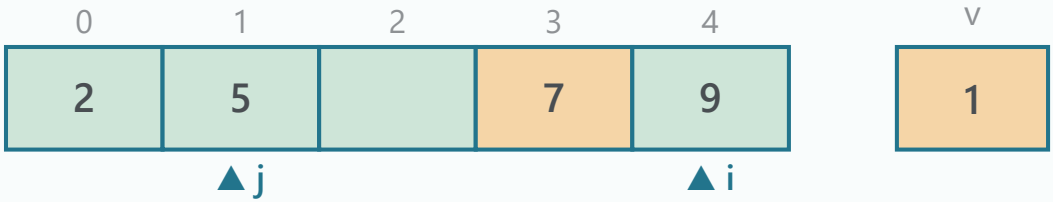
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // shift right
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 4` `v = 1` `j = 2`

`a = [2, 5, 7, 9, 9]`

Code Trace: `insertion(a, 0, 4)`



a[2] = 7 > v = 1 → 7 shifts one right (a[3]); j = 1.

Step 12 / 15

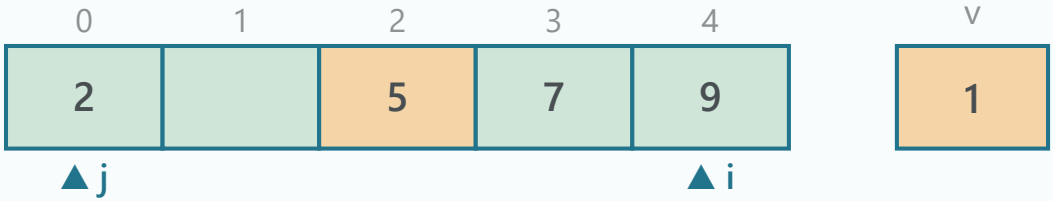
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // shift right
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 4` `v = 1` `j = 1`

`a = [2, 5, 7, 7, 9]`

Code Trace: `insertion(a, 0, 4)`



a[1] = 5 > v = 1 → 5 shifts one right (a[2]); j = 0.

Step 13 / 15

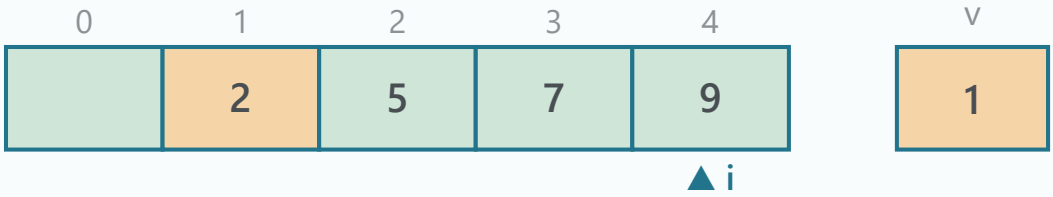
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // shift right
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 4` `v = 1` `j = 0`

`a = [2, 5, 5, 7, 9]`

Code Trace: `insertion(a, 0, 4)`



a[0] = 2 > v = 1 → 2 shifts one right (a[1]); j = -1.

Step 14 / 15

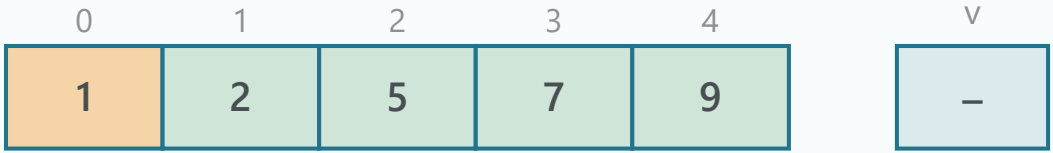
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
▶              a[j + 1] = a[j]; // shift right
▶              j--;
8          }
9          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 4` `v = 1` `j = -1`

`a = [2, 2, 5, 7, 9]`

Code Trace: `insertion(a, 0, 4)`



j = −1, the start of the array → the loop ends; **a[0] = 1**. The sorted prefix is now a[0..4].

Step 15 / 15

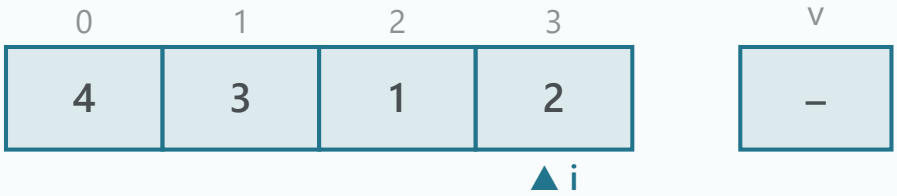
```
1  void insertion(int a[], int L, int R) {
2      for (int i = L + 1; i <= R; i++) {
3          int v = a[i];       // to insert
4          int j = i - 1;      // a[L..i-1] sorted
▶          while (j >= L && a[j] > v) {
6              a[j + 1] = a[j]; // shift right
7              j--;
8          }
▶          a[j + 1] = v;
10     }
11 }
```

VARIABLES

`i = 4` `v = 1` `j = -1`

`a = [1, 2, 5, 7, 9]`

Code Trace: `insertion1(a, 0, 3)` (with a Sentinel)



a[3] = 2 ≥ a[2] = 1 → no swap. The minimum moves left.

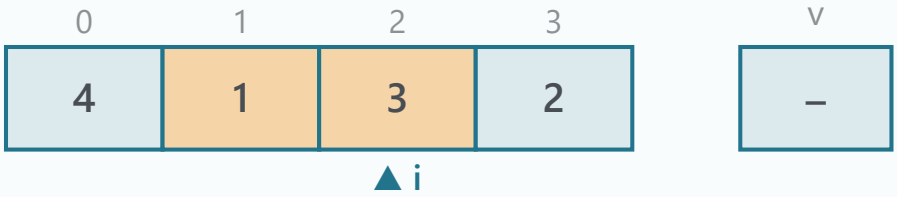
Step 1 / 10

```
1  void insertion1(int a[], int L, int R) {
▶      for (int i = R; i > L; i--) // sentinel
▶          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
6          while (less(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
10         a[j] = v;
11     }
12 }
```

VARIABLES

`i = 3` `a = [4, 3, 1, 2]`

Code Trace: `insertion1(a, 0, 3)` (with a Sentinel)



a[2] = 1 < a[1] = 3 → swap. The minimum moves left.

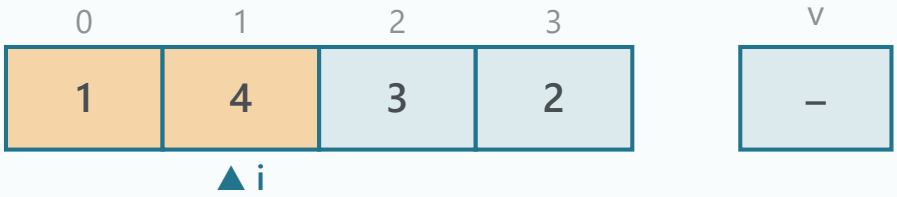
Step 2 / 10

```
1  void insertion1(int a[], int L, int R) {
▶      for (int i = R; i > L; i--) // sentinel
▶          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
6          while (less(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
10         a[j] = v;
11     }
12 }
```

VARIABLES

`i = 2` `a = [4, 1, 3, 2]`

Code Trace: `insertion1(a, 0, 3)` (with a Sentinel)



a[1] = 1 < a[0] = 4 → swap. **a[0] = 1** is now the minimum: the sentinel is ready.

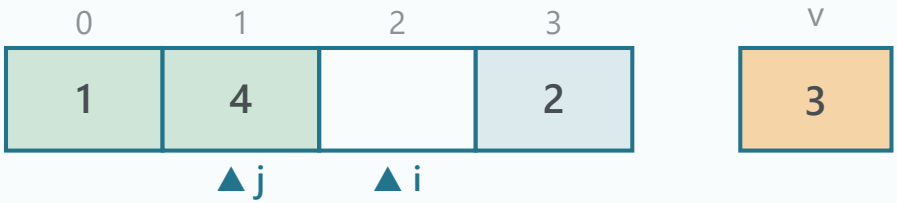
Step 3 / 10

```
1  void insertion1(int a[], int L, int R) {
▶      for (int i = R; i > L; i--) // sentinel
▶          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
6          while (less(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
10         a[j] = v;
11     }
12 }
```

VARIABLES

`i = 1` `a = [1, 4, 3, 2]`

Code Trace: `insertion1(a, 0, 3)` (with a Sentinel)



i = 2: v = 3. a[0..1] is sorted.

Step 4 / 10

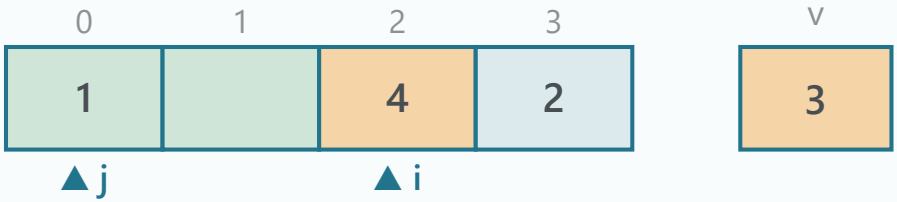
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // sentinel
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
▶          int v = a[i], j = i;
6          while (less(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
10         a[j] = v;
11     }
12 }
```

VARIABLES

i = 2 v = 3 j = 2

a = [1, 4, 3, 2]

Code Trace: `insertion1(a, 0, 3)` (with a Sentinel)



v = 3 < a[1] = 4 → shift; j = 1. No bounds test: the sentinel a[0] will stop it.

Step 5 / 10

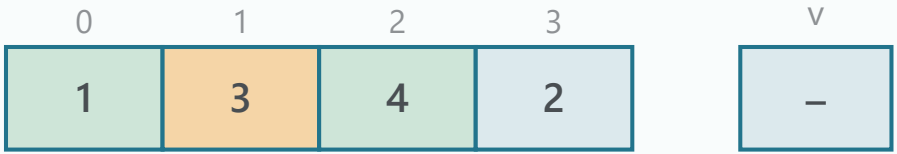
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // sentinel
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
▶          while (less(v, a[j - 1])) {
▶              a[j] = a[j - 1];
▶              j--;
9          }
10         a[j] = v;
11     }
12 }
```

VARIABLES

i = 2 | v = 3 | j = 1

a = [1, 4, 4, 2]

Code Trace: `insertion1(a, 0, 3)` (with a Sentinel)



a[0] = 1 ≤ v → stop; **a[1] = 3**.

Step 6 / 10

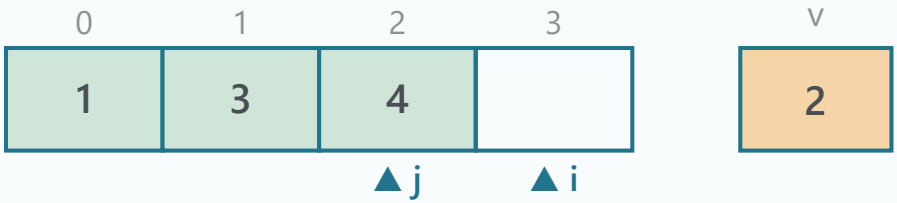
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // sentinel
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
▶          while (less(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
▶          a[j] = v;
11     }
12 }
```

VARIABLES

`i = 2` `v = 3` `j = 1`

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

Code Trace: `insertion1(a, 0, 3)` (with a Sentinel)



i = 3: v = 2. a[0..2] is sorted.

Step 7 / 10

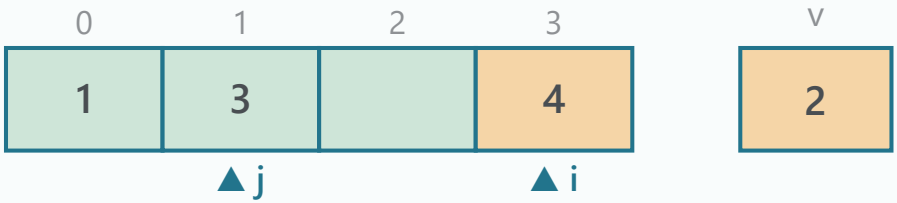
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // sentinel
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
▶          int v = a[i], j = i;
6          while (less(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
10         a[j] = v;
11     }
12 }
```

VARIABLES

i = 3 v = 2 j = 3

a = [1, 3, 4, 2]

Code Trace: `insertion1(a, 0, 3)` (with a Sentinel)



v = 2 < a[2] = 4 → shift; j = 2. No bounds test: the sentinel a[0] will stop it.

Step 8 / 10

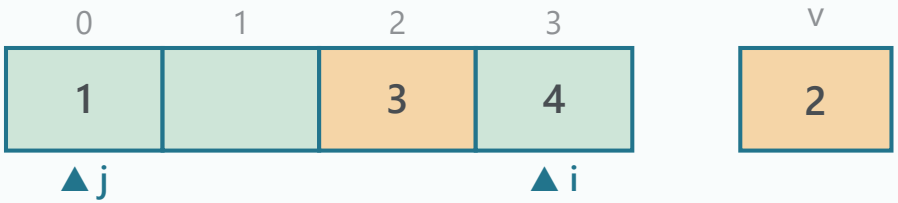
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // sentinel
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
▶          while (less(v, a[j - 1])) {
▶              a[j] = a[j - 1];
▶              j--;
9          }
10         a[j] = v;
11     }
12 }
```

VARIABLES

i = 3 | v = 2 | j = 2

a = [1, 3, 4, 4]

Code Trace: `insertion1(a, 0, 3)` (with a Sentinel)



v = 2 < a[1] = 3 → shift; j = 1. No bounds test: the sentinel a[0] will stop it.

Step 9 / 10

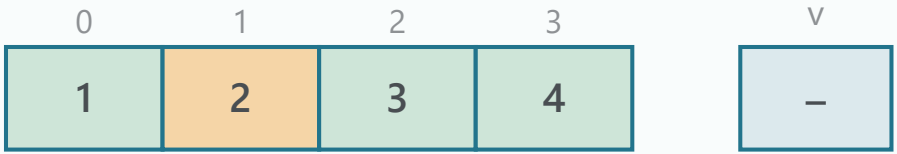
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // sentinel
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
▶          while (less(v, a[j - 1])) {
▶              a[j] = a[j - 1];
▶              j--;
9          }
10         a[j] = v;
11     }
12 }
```

VARIABLES

i = 3 | v = 2 | j = 1

a = [1, 3, 3, 4]

Code Trace: `insertion1(a, 0, 3)` (with a Sentinel)



a[0] = 1 ≤ v → stop; **a[1] = 2**.

Step 10 / 10

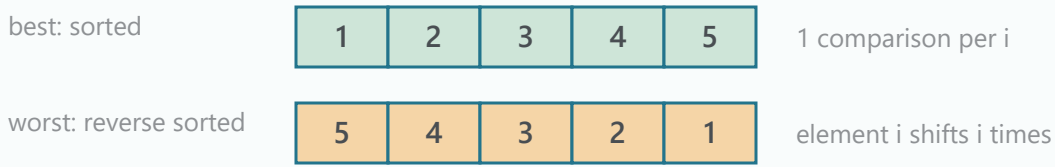
```
1  void insertion1(int a[], int L, int R) {
2      for (int i = R; i > L; i--) // sentinel
3          if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
4      for (int i = L + 2; i <= R; i++) {
5          int v = a[i], j = i;
▶          while (less(v, a[j - 1])) {
7              a[j] = a[j - 1];
8              j--;
9          }
▶          a[j] = v;
11     }
12 }
```

VARIABLES

i = 3 v = 2 j = 1

a = [1, 2, 3, 4]

Insertion Sort: Best and Worst Case



Best: Ω(n)

On sorted input, each element is compared only with the rightmost value of the sorted list; it is larger, so it stays put: n − 1 comparisons in total.

Worst: $O(n^2)$

On reverse-sorted input, each element is compared with the whole sorted list and inserted at the far left; the sorted part shifts right every pass: 1 + 2 + … + (n−1) = n(n−1)/2.

Insertion sort is **stable** (`a[j] > v` is strict, so equal elements are not passed) and in place. On nearly sorted data it is almost linear.

Average-Case Analysis: The Possible Cases for j = 3

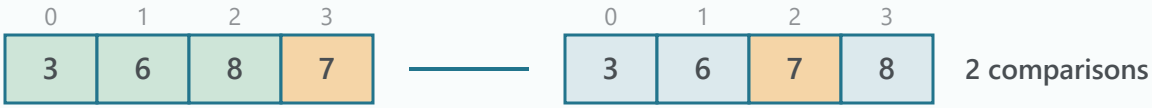


Case 1: 8 > 10 is false → **1 comparison**.

Step 1 / 4

Element j lands in one of j + 1 positions: 1 comparison in 1 case, 2 in 1 case, **3 in 2 cases**. The cases are equally likely; only the last count, “3 comparisons”, occurs twice. Average: (1 + 2 + 3 + 3)/4 = **2.25**.

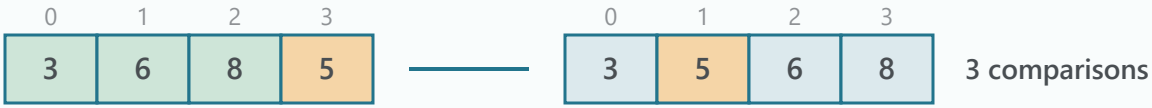
Average-Case Analysis: The Possible Cases for j = 3



Case 2: 8 > 7 true, 6 > 7 false → **2 comparisons**.

Element j lands in one of j + 1 positions: 1 comparison in 1 case, 2 in 1 case, **3 in 2 cases**. The cases are equally likely; only the last count, “3 comparisons”, occurs twice. Average: (1 + 2 + 3 + 3)/4 = **2.25**.

Average-Case Analysis: The Possible Cases for j = 3



Case 3: 8 > 5 true, 6 > 5 true, 3 > 5 false → **3 comparisons**.

Element j lands in one of j + 1 positions: 1 comparison in 1 case, 2 in 1 case, **3 in 2 cases**. The cases are equally likely; only the last count, “3 comparisons”, occurs twice. Average: (1 + 2 + 3 + 3)/4 = **2.25**.

Average-Case Analysis: The Possible Cases for j = 3



at j = −1, j ≥ 0 is false: this test is not counted

Case 4: 8 > 2, 6 > 2, 3 > 2 all true; at j = −1, `j ≥ 0` is false and `a[j] > v` is never tested, so it costs **the same** as the previous case → **3 comparisons**.

Element j lands in one of j + 1 positions: 1 comparison in 1 case, 2 in 1 case, **3 in 2 cases**. The cases are equally likely; only the last count, “3 comparisons”, occurs twice. Average: (1 + 2 + 3 + 3)/4 = **2.25**.

Average Case: The Expected Number of Comparisons

$$E(C_j) = (1/(j+1)) \cdot [2j + \Sigma_{k=1}^{j-1} k] = (j+2)/2 - 1/(j+1)$$

$$E(Y) = \Sigma_{j=1}^{N-1} E(C_j) = (N^2 + 3N)/4 - H_N \approx N^2/4$$

- C_j, the number of comparisons to insert element j, is a **random variable**; $E(C_j)$ is its expected value.
- $H_N = 1 + 1/2 + \ldots + 1/N$ is the harmonic number: $H_N = \ln N + O(1)$.
- On average $\approx N^2/4$ comparisons and $\approx N^2/4$ **half-exchanges** (shifts, not swaps) → **$O(n^2)$**.

N	N!	all permutations	formula	$N^2/4$
3	6	2.667	2.667	2.25
4	24	4.917	4.917	4.00
5	120	7.717	7.717	6.25
6	720	11.050	11.050	9.00
7	5040	14.907	14.907	12.25

Table: all N! orderings were counted in Python; they match the formula exactly.

INSTRUCTOR "We know the best and the worst case; if we average the two, have we analyzed the average case? No! Average-case analysis has to look at every case in the sample space."

Bubble Sort

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

INSTRUCTOR “It can be coded more carefully so that it exits at once if the array is sorted. If a whole pass makes no swaps, there is no need for the next pass.”



Start: [7, 2, 9, 5, 1]. Each pass compares neighbors from right to left.

Step 1 / 5

Bubble Sort

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

INSTRUCTOR “It can be coded more carefully so that it exits at once if the array is sorted. If a whole pass makes no swaps, there is no need for the next pass.”



Pass i = 0: the smallest remaining value **1** bubbles up to a[0] → [1, 7, 2, 9, 5].

Bubble Sort

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

INSTRUCTOR “It can be coded more carefully so that it exits at once if the array is sorted. If a whole pass makes no swaps, there is no need for the next pass.”

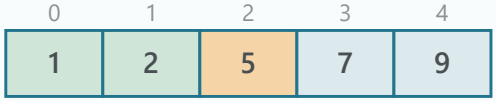


Pass i = 1: the smallest remaining value **2** bubbles up to a[1] → [1, 2, 7, 5, 9].

Bubble Sort

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

INSTRUCTOR “It can be coded more carefully so that it exits at once if the array is sorted. If a whole pass makes no swaps, there is no need for the next pass.”

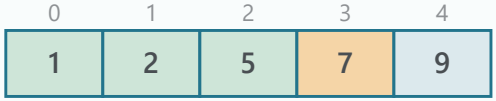


Pass i = 2: the smallest remaining value **5** bubbles up to a[2] → [1, 2, 5, 7, 9].

Bubble Sort

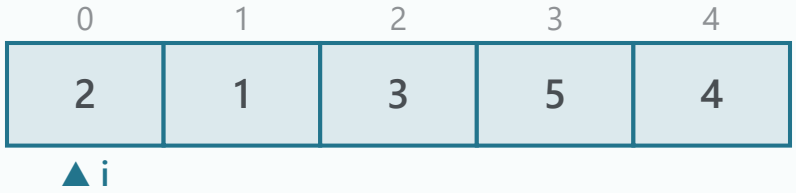
```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

INSTRUCTOR “It can be coded more carefully so that it exits at once if the array is sorted. If a whole pass makes no swaps, there is no need for the next pass.”



Pass i = 3: no swap; the smallest remaining value **7** is already in a[3] → [1, 2, 5, 7, 9].

Code Trace: `bubble2(a, 0, 4)` with Early Exit



Pass i = 0 begins; the swap flag is cleared.

Step 1 / 11

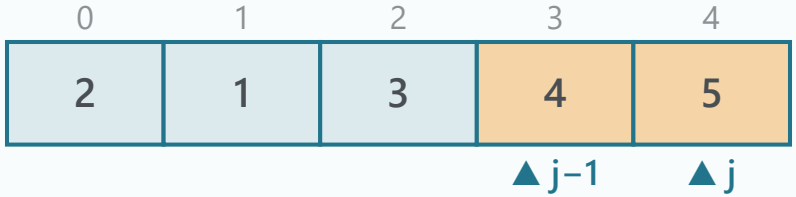
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
▶          int swapped = 0;     // any swap this pass?
4          for (int j = R; j > i; j--)
5              if (less(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  swapped = 1;
8              }
9          if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 0` `swapped = 0`

`a = [2, 1, 3, 5, 4]`

Code Trace: `bubble2(a, 0, 4)` with Early Exit



a[4] < a[3] → swap: [2, 1, 3, 4, 5]; **swapped = 1**.

Step 2 / 11

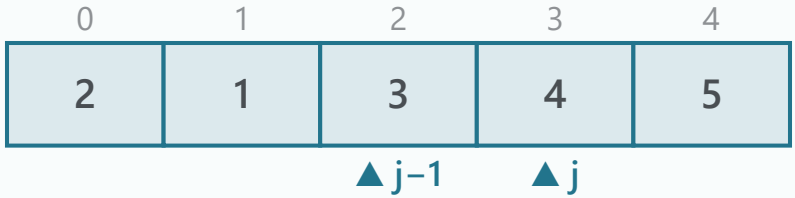
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int swapped = 0;     // any swap this pass?
4          for (int j = R; j > i; j--)
▶              if (less(a[j], a[j - 1])) {
▶                  SWAP(a[j], a[j - 1]);
▶                  swapped = 1;
8              }
9          if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 0` `j = 4` `swapped = 1`

`a = [2, 1, 3, 4, 5]`

Code Trace: `bubble2(a, 0, 4)` with Early Exit



a[3] = 4 ≥ a[2] = 3 → no swap.

Step 3 / 11

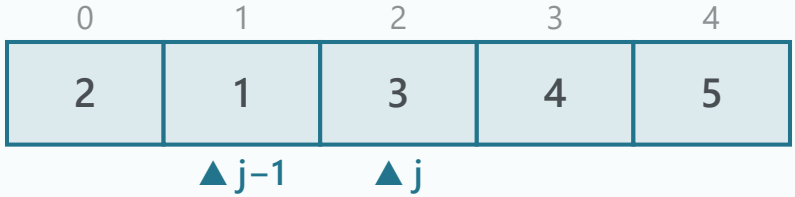
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int swapped = 0;     // any swap this pass?
4          for (int j = R; j > i; j--)
▶              if (less(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  swapped = 1;
8              }
9          if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 0` `j = 3` `swapped = 1`

`a = [2, 1, 3, 4, 5]`

Code Trace: `bubble2(a, 0, 4)` with Early Exit



a[2] = 3 ≥ a[1] = 1 → no swap.

Step 4 / 11

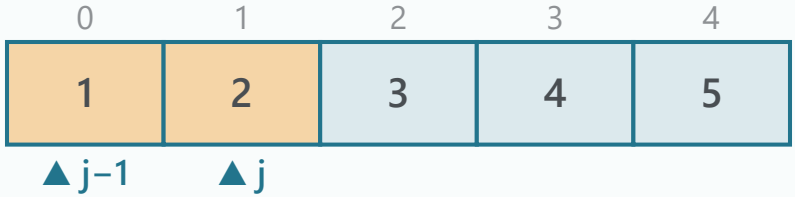
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int swapped = 0;     // any swap this pass?
4          for (int j = R; j > i; j--)
▶              if (less(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  swapped = 1;
8              }
9          if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 0` `j = 2` `swapped = 1`

`a = [2, 1, 3, 4, 5]`

Code Trace: `bubble2(a, 0, 4)` with Early Exit



a[1] < a[0] → swap: [1, 2, 3, 4, 5]; **swapped = 1**.

Step 5 / 11

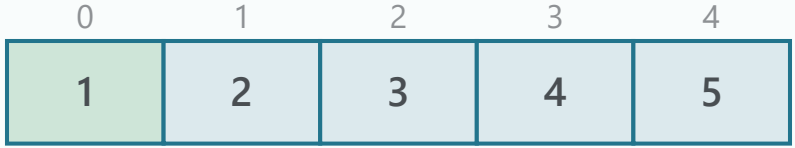
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int swapped = 0;     // any swap this pass?
4          for (int j = R; j > i; j--)
▶              if (less(a[j], a[j - 1])) {
▶                  SWAP(a[j], a[j - 1]);
▶                  swapped = 1;
8              }
9          if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 0` `j = 1` `swapped = 1`

`a = [1, 2, 3, 4, 5]`

Code Trace: `bubble2(a, 0, 4)` with Early Exit



swapped = 1 → the array cannot be assumed sorted yet; go to the next pass.

Step 6 / 11

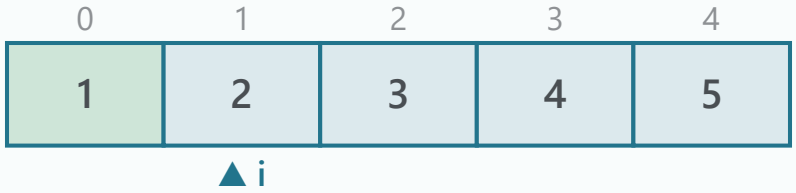
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int swapped = 0;     // any swap this pass?
4          for (int j = R; j > i; j--)
5              if (less(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  swapped = 1;
8              }
▶          if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 0` `swapped = 1`

`a = [1, 2, 3, 4, 5]`

Code Trace: `bubble2(a, 0, 4)` with Early Exit



Pass i = 1 begins; the swap flag is cleared.

Step 7 / 11

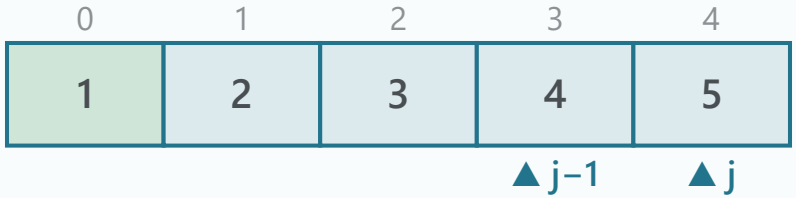
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
▶          int swapped = 0;     // any swap this pass?
4          for (int j = R; j > i; j--)
5              if (less(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  swapped = 1;
8              }
9          if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 1` `swapped = 0`

`a = [1, 2, 3, 4, 5]`

Code Trace: `bubble2(a, 0, 4)` with Early Exit



a[4] = 5 ≥ a[3] = 4 → no swap.

Step 8 / 11

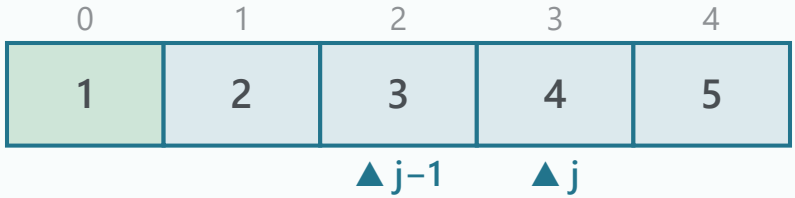
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int swapped = 0;     // any swap this pass?
4          for (int j = R; j > i; j--)
▶              if (less(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  swapped = 1;
8              }
9          if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 1` `j = 4` `swapped = 0`

`a = [1, 2, 3, 4, 5]`

Code Trace: `bubble2(a, 0, 4)` with Early Exit



a[3] = 4 ≥ a[2] = 3 → no swap.

Step 9 / 11

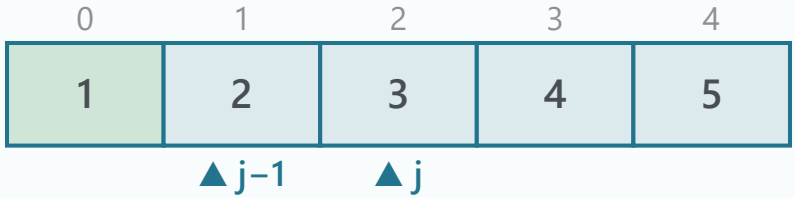
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int swapped = 0;     // any swap this pass?
4          for (int j = R; j > i; j--)
▶              if (less(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  swapped = 1;
8              }
9          if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 1` `j = 3` `swapped = 0`

`a = [1, 2, 3, 4, 5]`

Code Trace: `bubble2(a, 0, 4)` with Early Exit



a[2] = 3 ≥ a[1] = 2 → no swap.

Step 10 / 11

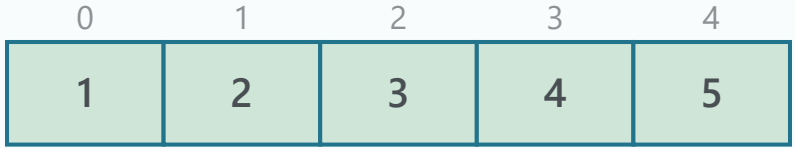
```
1  void bubble2(int a[], int L, int R) {
2      for (int i = L; i < R; i++) {
3          int swapped = 0;     // any swap this pass?
4          for (int j = R; j > i; j--)
▶              if (less(a[j], a[j - 1])) {
6                  SWAP(a[j], a[j - 1]);
7                  swapped = 1;
8              }
9          if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 1` `j = 2` `swapped = 0`

`a = [1, 2, 3, 4, 5]`

Code Trace: `bubble2(a, 0, 4)` with Early Exit



No swap in this pass → the array is sorted; **early exit**. Two useless passes were skipped.

Step 11 / 11

```
 1 void bubble2(int a[], int L, int R) {
 2     for (int i = L; i < R; i++) {
 3         int swapped = 0;     // any swap this pass?
 4         for (int j = R; j > i; j--)
 5             if (less(a[j], a[j - 1])) {
 6                 SWAP(a[j], a[j - 1]);
 7                 swapped = 1;
 8             }
 ▶         if (!swapped) return; // sorted: exit early
10     }
11 }
```

VARIABLES

`i = 1` `swapped = 0`

`a = [1, 2, 3, 4, 5]`

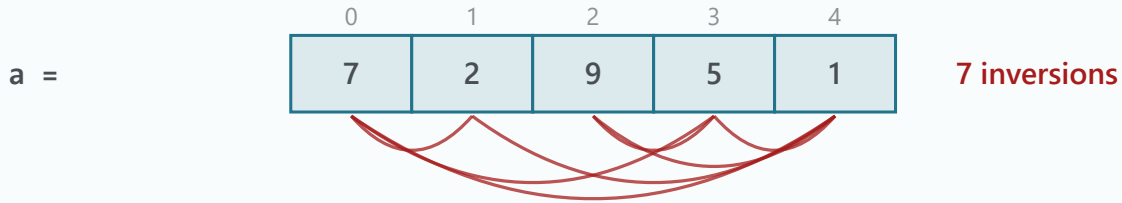
Analysis of Bubble Sort

- Comparisons: $n(n-1)/2 \approx \mathbf{n^2/2}$ in every case (without early exit).
- Exchanges: $\mathbf{n^2/2}$ in the worst case (reverse sorted); on average as many as the **inversions**, $\approx \mathbf{n^2/4}$.
- Complexity $\mathbf{O(n^2)}$; the early-exit version is **O(n)** on sorted input.
- Only neighbors are swapped and equal elements never are → **stable**, in place.

n	comp.	swaps (worst)	swaps (avg.)
4	6	6	3
5	10	10	5
6	15	15	7.5
7	21	21	10.5

Average column: all n! orderings were counted. Some sources (e.g. Sedgewick) also give $n^2/2$ exchanges on average; the count shows the average is $n^2/4$.

Inversions: a Common Measure



Inversion: a pair with i < j and a[i] > a[j]. [7, 2, 9, 5, 1] → (7,2), (7,5), (7,1), (2,1), (9,5), (9,1), (5,1)

- A neighbor swap (bubble) or a one-position shift (insertion) removes exactly **one** inversion.
- So in both, the number of swaps/shifts = the number of inversions: 7 for this array (the traces also made 7 shifts).
- A reverse-sorted array has n(n−1)/2 inversions; a random one has **n(n−1)/4** on average (each pair is inverted with probability ½).
- Every algorithm that works on neighbors is $\Omega(n^2)$ on average: Shell sort and quicksort, with long-distance swaps, break through this wall.

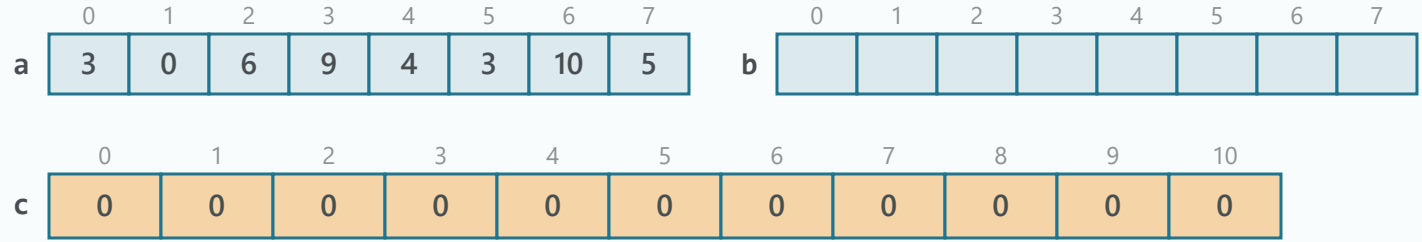
Counting Sort

```c
int counting_sort(const int a[], int b[],
                  int r, int k) {
    int *c = malloc((k + 1) * sizeof *c);
    if (c == NULL) return -1;
    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
    for (int i = 1; i <= k; i++)             // L3
        c[i] += c[i - 1];
    for (int j = r; j >= 0; j--) {           // L4
        b[c[a[j]] - 1] = a[j];
        c[a[j]]--;
    }
    free(c);
    return 0;
}
```

name	meaning
a[]	input array
b[]	sorted output array
c[]	auxiliary count array (k+1 elements)
r	a[0..r] is sorted (r+1 elements)
k	values lie in 0…k

- **Harold H. Seward**, 1954. Sorts integers from a known range.
- Values are never **compared** with each other → a *non-comparison sort* (bucket sort is in this class too).
- `int c[K]` needs a compile-time constant size; here we use `malloc` + a check + `free`.

Code Trace: `counting_sort`, Counting (L1–L3)



L1: c[0..10] is cleared. Here we will count how many times each value occurs.

Step 1 / 19

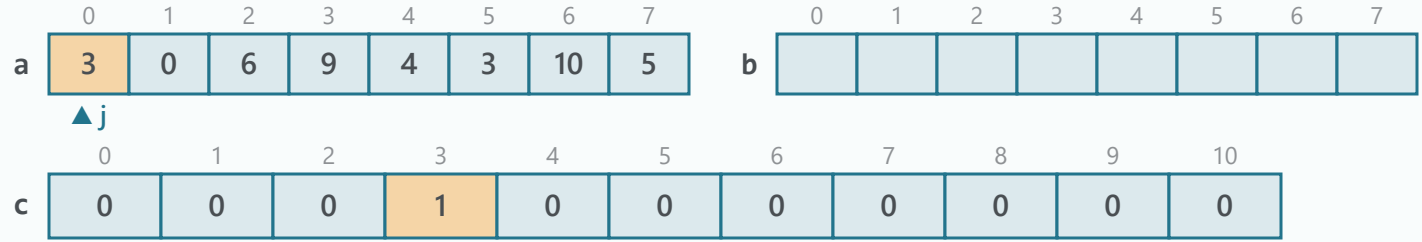
```
▶   for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2   for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3   for (int i = 1; i <= k; i++)            // L3
4       c[i] += c[i - 1];
```

VARIABLES

`r = 7` `k = 10`

`c = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`

Code Trace: `counting_sort`, Counting (L1–L3)



L2: a[0] = 3 → **c[3]++** = 1.

Step 2 / 19

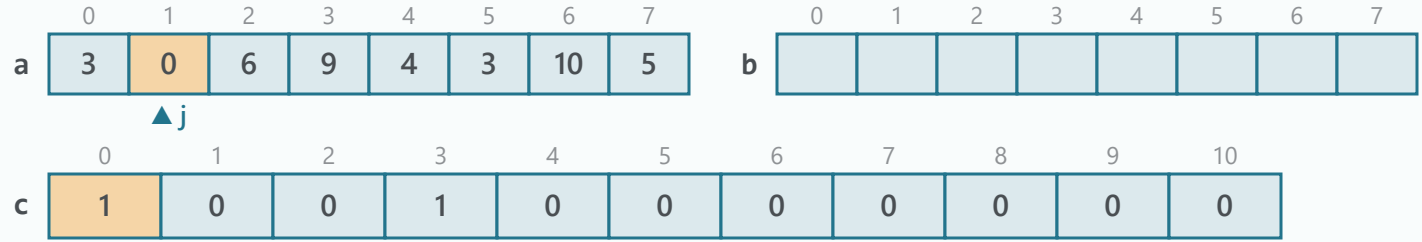
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

VARIABLES

`j = 0` `a[j] = 3`

`c = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`

Code Trace: `counting_sort`, Counting (L1–L3)



L2: a[1] = 0 → **c[0]++** = 1.

Step 3 / 19

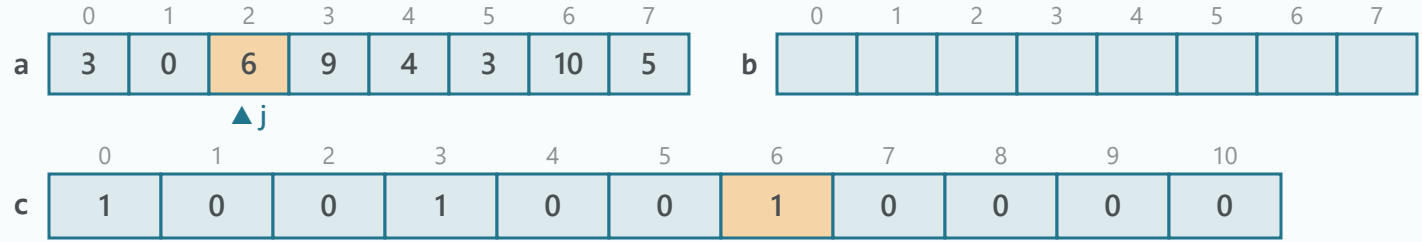
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

VARIABLES

`j = 1` `a[j] = 0`

`c = [1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`

Code Trace: `counting_sort`, Counting (L1–L3)



L2: a[2] = 6 → **c[6]++** = 1.

Step 4 / 19

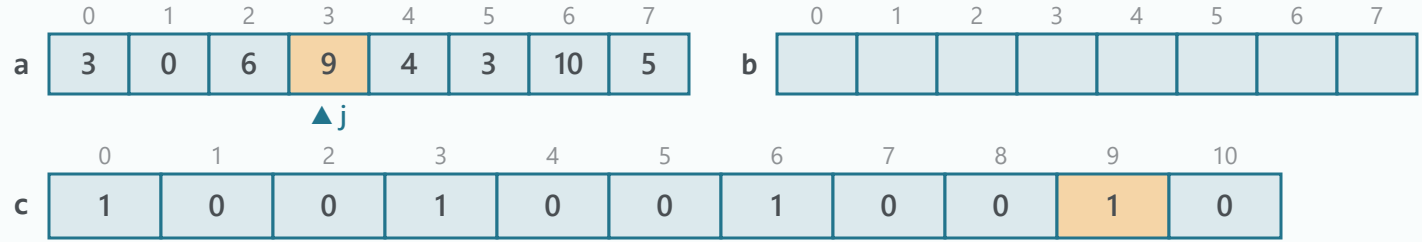
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

VARIABLES

`j = 2` `a[j] = 6`

`c = [1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0]`

Code Trace: `counting_sort`, Counting (L1–L3)



L2: a[3] = 9 → **c[9]++** = 1.

Step 5 / 19

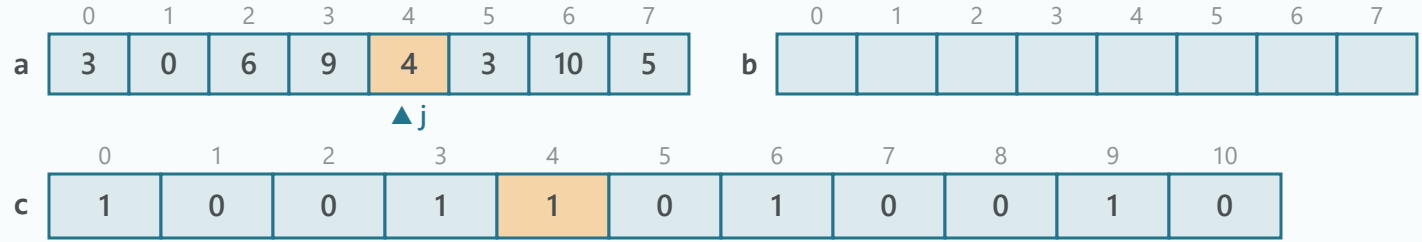
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

VARIABLES

`j = 3` `a[j] = 9`

`c = [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]`

Code Trace: `counting_sort`, Counting (L1–L3)



L2: a[4] = 4 → **c[4]++** = 1.

Step 6 / 19

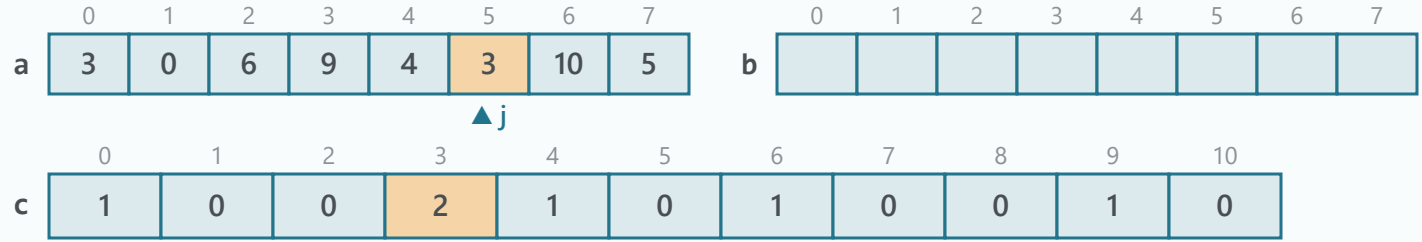
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

VARIABLES

`j = 4` `a[j] = 4`

`c = [1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0]`

Code Trace: `counting_sort`, Counting (L1–L3)



L2: a[5] = 3 → **c[3]++** = 2. (A second 3!)

Step 7 / 19

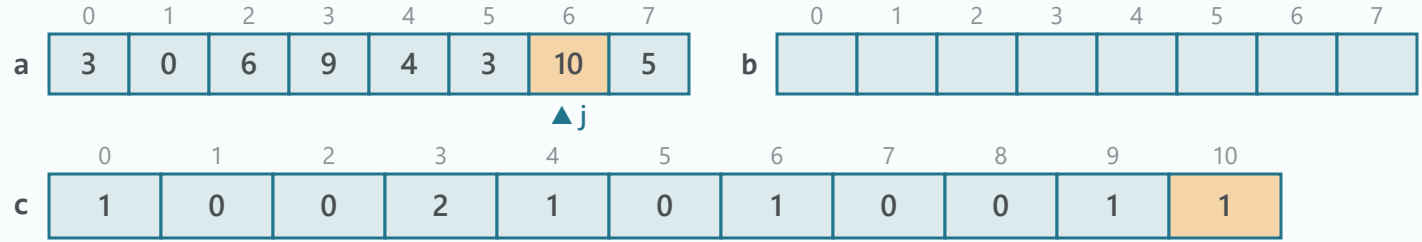
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

VARIABLES

`j = 5` `a[j] = 3`

`c = [1, 0, 0, 2, 1, 0, 1, 0, 0, 1, 0]`

Code Trace: `counting_sort`, Counting (L1–L3)



L2: a[6] = 10 → **c[10]++** = 1.

Step 8 / 19

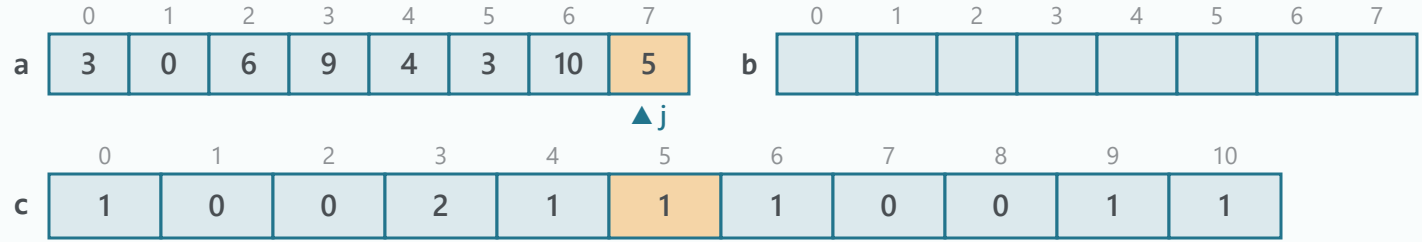
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

VARIABLES

`j = 6` `a[j] = 10`

`c = [1, 0, 0, 2, 1, 0, 1, 0, 0, 1, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L2: a[7] = 5 → **c[5]++** = 1.

Step 9 / 19

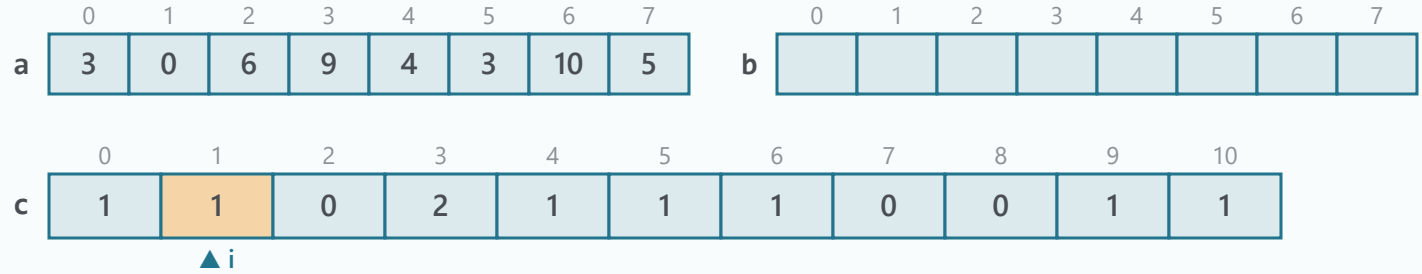
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
▶     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
3     for (int i = 1; i <= k; i++)             // L3
4         c[i] += c[i - 1];
```

VARIABLES

`j = 7` `a[j] = 5`

`c = [1, 0, 0, 2, 1, 1, 1, 0, 0, 1, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L3: c[1] += c[0] → 1: “1 element is ≤ 1”.

Step 10 / 19

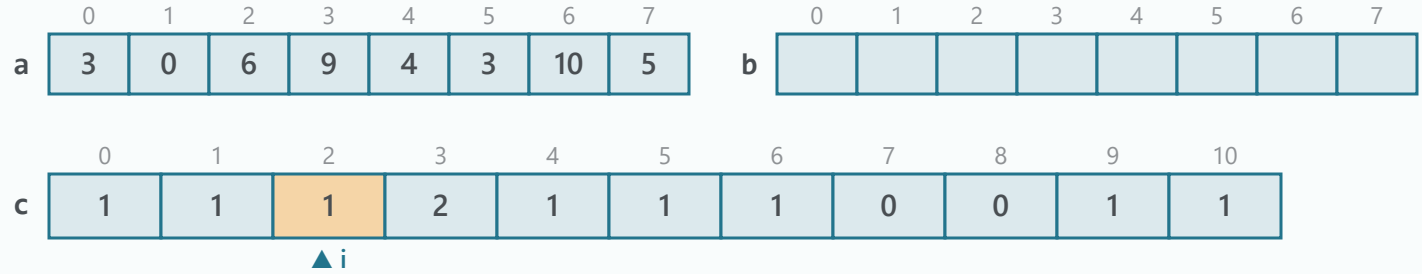
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶     for (int i = 1; i <= k; i++)            // L3
▶         c[i] += c[i - 1];
```

VARIABLES

`i = 1`

`c = [1, 1, 0, 2, 1, 1, 1, 0, 0, 1, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L3: c[2] += c[1] → 1: “1 element is ≤ 2”.

Step 11 / 19

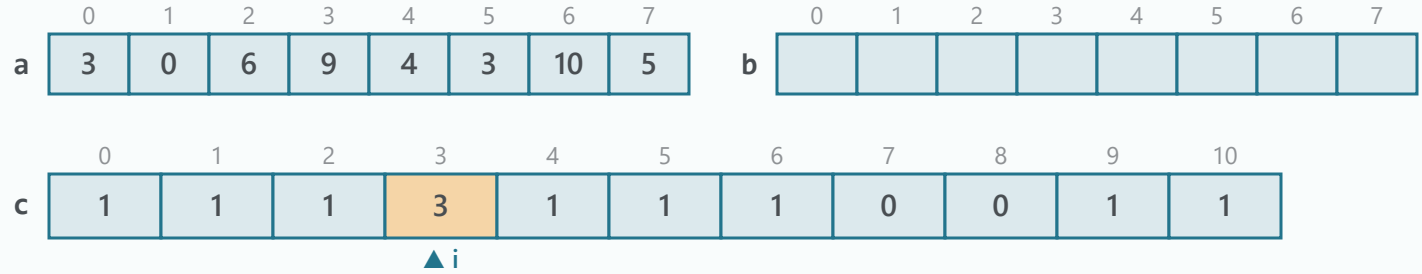
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶     for (int i = 1; i <= k; i++)            // L3
▶         c[i] += c[i - 1];
```

VARIABLES

`i = 2`

`c = [1, 1, 1, 2, 1, 1, 1, 0, 0, 1, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L3: c[3] += c[2] → 3: “3 elements are ≤ 3”. A 3 goes to position 2.

Step 12 / 19

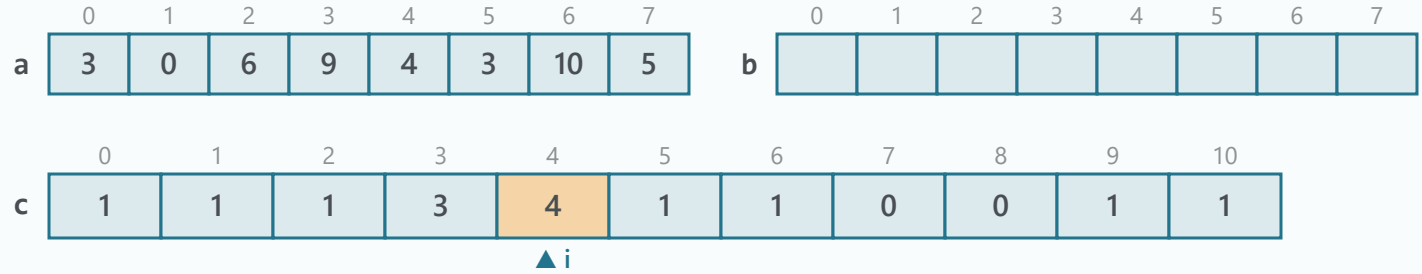
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶     for (int i = 1; i <= k; i++)            // L3
▶         c[i] += c[i - 1];
```

VARIABLES

`i = 3`

`c = [1, 1, 1, 3, 1, 1, 1, 0, 0, 1, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L3: c[4] += c[3] → 4: “4 elements are ≤ 4”. A 4 goes to position 3.

Step 13 / 19

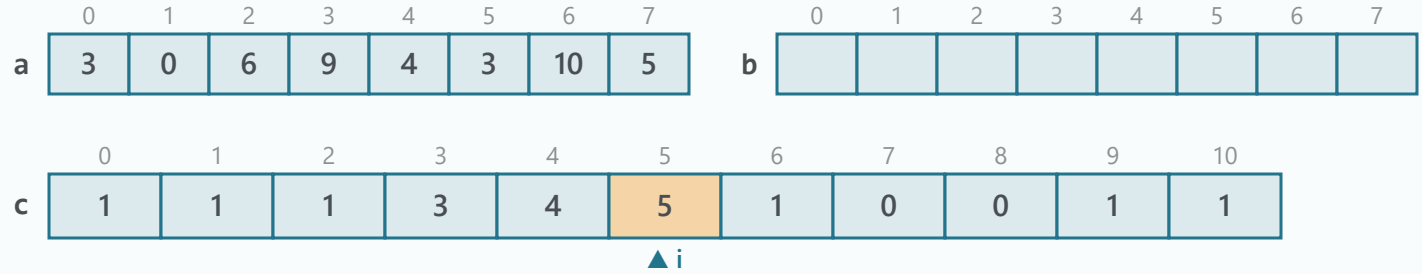
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶     for (int i = 1; i <= k; i++)            // L3
▶         c[i] += c[i - 1];
```

VARIABLES

`i = 4`

`c = [1, 1, 1, 3, 4, 1, 1, 0, 0, 1, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L3: c[5] += c[4] → 5: “5 elements are ≤ 5”. A 5 goes to position 4.

Step 14 / 19

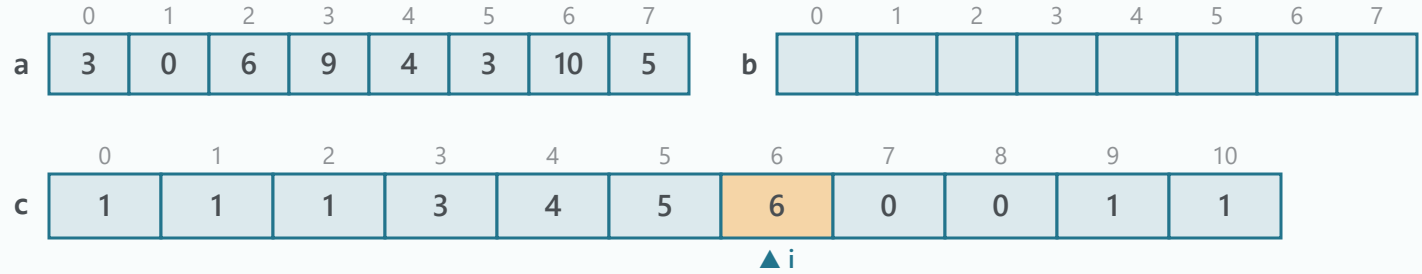
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶     for (int i = 1; i <= k; i++)            // L3
▶         c[i] += c[i - 1];
```

VARIABLES

`i = 5`

`c = [1, 1, 1, 3, 4, 5, 1, 0, 0, 1, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L3: c[6] += c[5] → 6: “6 elements are ≤ 6”. A 6 goes to position 5.

Step 15 / 19

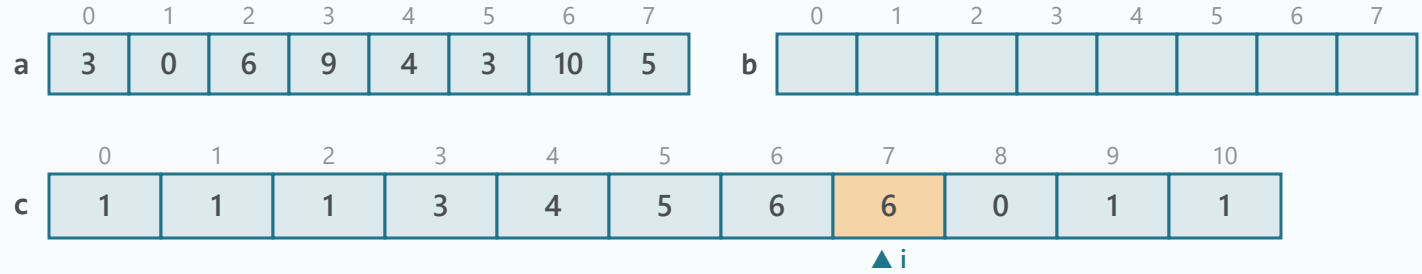
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

VARIABLES

`i = 6`

`c = [1, 1, 1, 3, 4, 5, 6, 0, 0, 1, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L3: c[7] += c[6] → 6: “6 elements are ≤ 7”.

Step 16 / 19

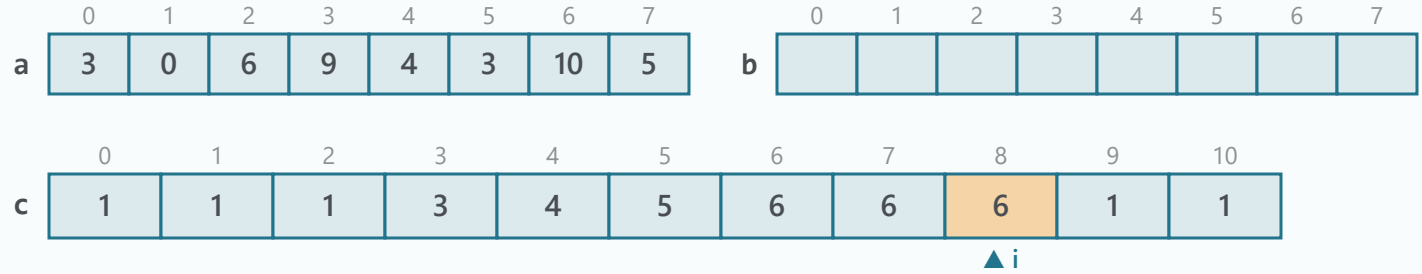
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

VARIABLES

`i = 7`

`c = [1, 1, 1, 3, 4, 5, 6, 6, 0, 1, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L3: c[8] += c[7] → 6: “6 elements are ≤ 8”.

Step 17 / 19

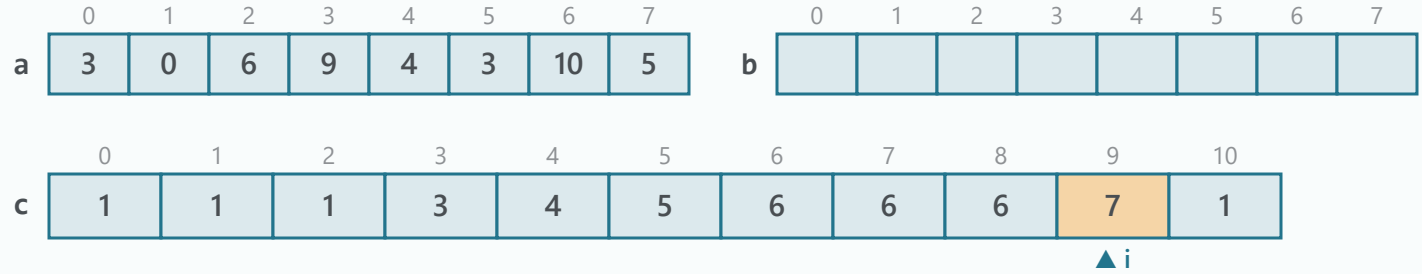
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

VARIABLES

`i = 8`

`c = [1, 1, 1, 3, 4, 5, 6, 6, 6, 1, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L3: c[9] += c[8] → 7: “7 elements are ≤ 9”. A 9 goes to position 6.

Step 18 / 19

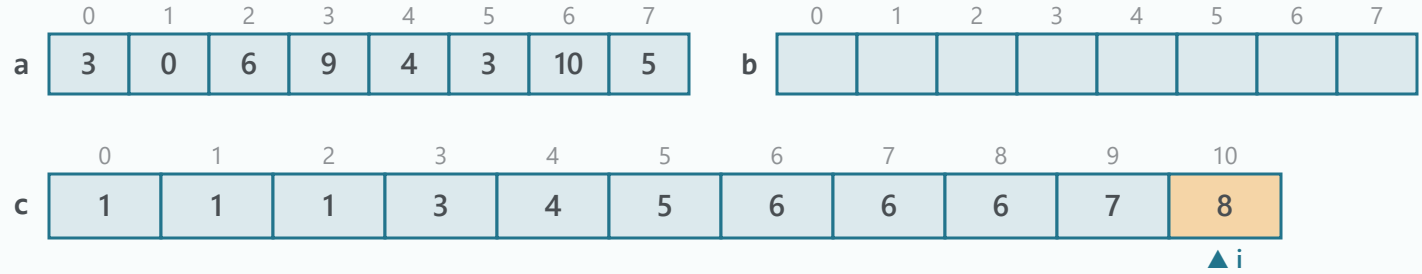
```
1     for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2     for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶     for (int i = 1; i <= k; i++)            // L3
▶         c[i] += c[i - 1];
```

VARIABLES

`i = 9`

`c = [1, 1, 1, 3, 4, 5, 6, 6, 6, 7, 1]`

Code Trace: `counting_sort`, Counting (L1–L3)



L3: c[10] += c[9] → 8: “8 elements are ≤ 10”. A 10 goes to position 7.

Step 19 / 19

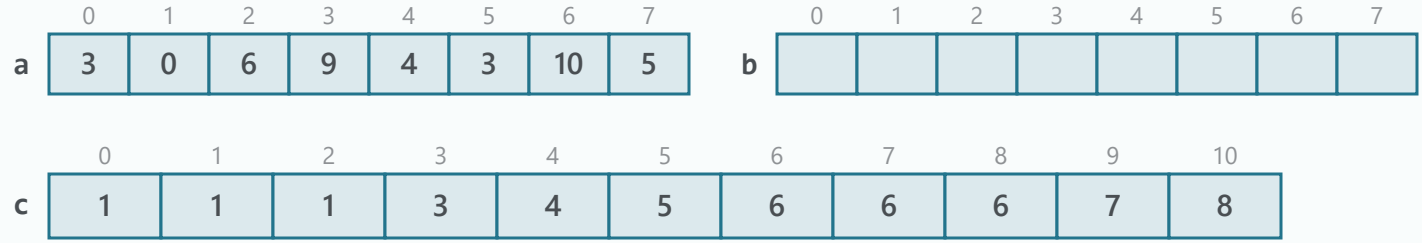
```
1    for (int i = 0; i <= k; i++) c[i] = 0;   // L1
2    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
▶    for (int i = 1; i <= k; i++)            // L3
▶        c[i] += c[i - 1];
```

VARIABLES

`i = 10`

`c = [1, 1, 1, 3, 4, 5, 6, 6, 6, 7, 8]`

Code Trace (Continued): Placing into b with L4



After L3: c[x] = “number of elements ≤ x”. E.g. c[9] = 7 → a 9 goes to position 7 − 1 = **6**.

Step 1 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
2        b[c[a[j]] - 1] = a[j];
3        c[a[j]]--;
4    }
```

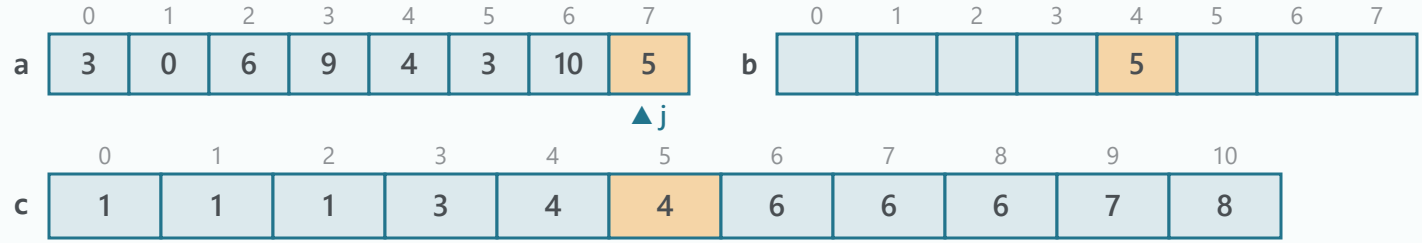
VARIABLES

`r = 7`

`c = [1, 1, 1, 3, 4, 5, 6, 6, 6, 7, 8]`

`b = [_, _, _, _, _, _, _, _]`

Code Trace (Continued): Placing into b with L4



L4: a[7] = 5, c[5] = 5 → **b[4] = 5**; c[5]-- = 4.

Step 2 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

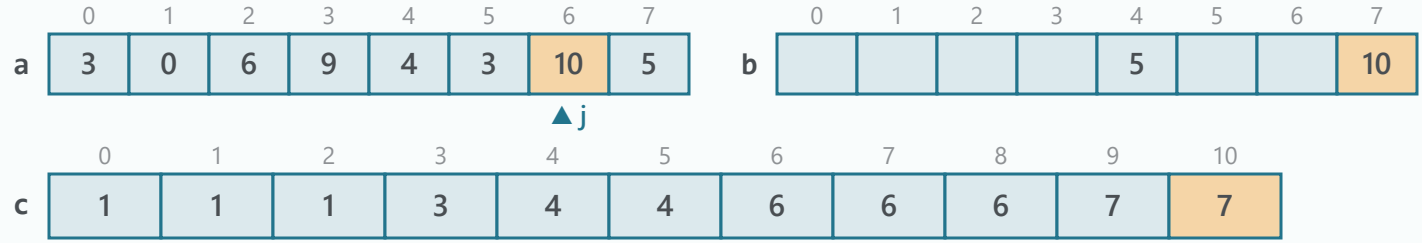
VARIABLES

`j = 7` `a[j] = 5`

`c = [1, 1, 1, 3, 4, 4, 6, 6, 6, 7, 8]`

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

Code Trace (Continued): Placing into b with L4



L4: a[6] = 10, c[10] = 8 → **b[7] = 10**; c[10]-- = 7.

Step 3 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

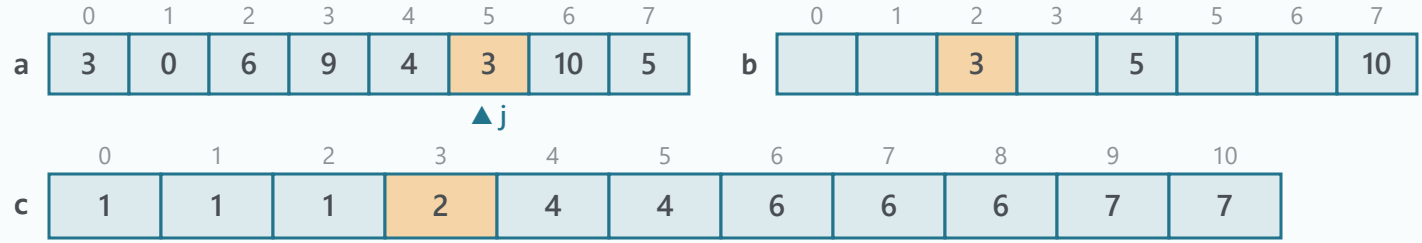
VARIABLES

`j = 6` `a[j] = 10`

`c = [1, 1, 1, 3, 4, 4, 6, 6, 6, 7, 7]`

`b = [_, _, _, _, 5, _, _, 10]`

Code Trace (Continued): Placing into b with L4



L4: a[5] = 3, c[3] = 3 → **b[2] = 3**; c[3]-- = 2 (the next equal value lands one to the left).

Step 4 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

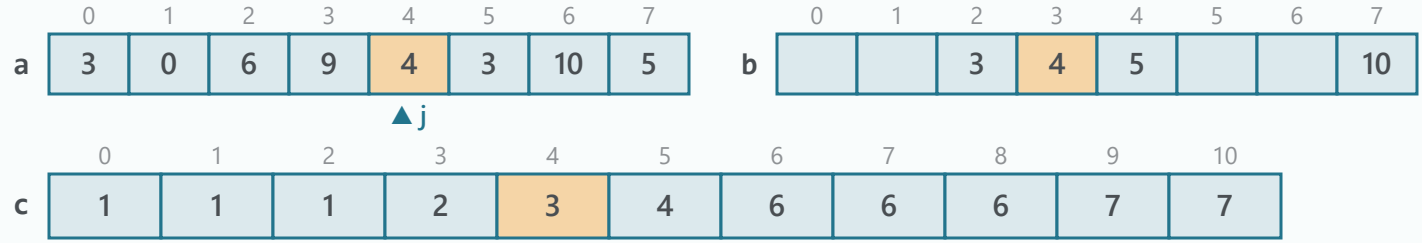
VARIABLES

`j = 5` `a[j] = 3`

`c = [1, 1, 1, 2, 4, 4, 6, 6, 6, 7, 7]`

`b = [_, _, 3, _, 5, _, _, 10]`

Code Trace (Continued): Placing into b with L4



L4: a[4] = 4, c[4] = 4 → **b[3] = 4**; c[4]-- = 3.

Step 5 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

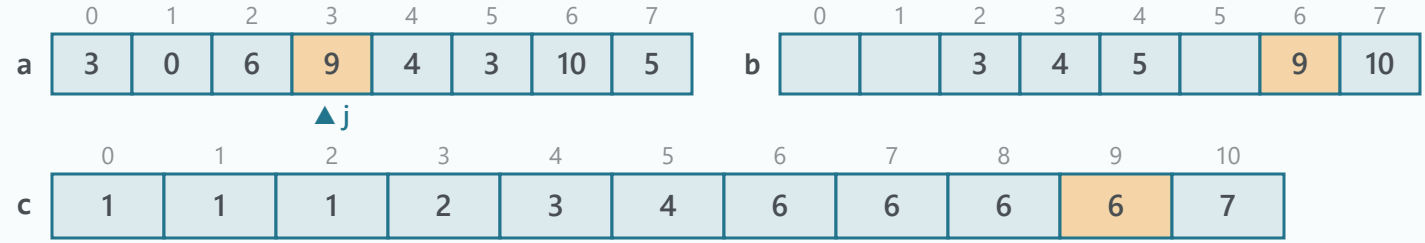
VARIABLES

`j = 4` `a[j] = 4`

`c = [1, 1, 1, 2, 3, 4, 6, 6, 6, 7, 7]`

`b = [_, _, 3, 4, 5, _, _, 10]`

Code Trace (Continued): Placing into b with L4



L4: a[3] = 9, c[9] = 7 → **b[6] = 9**; c[9]-- = 6.

Step 6 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

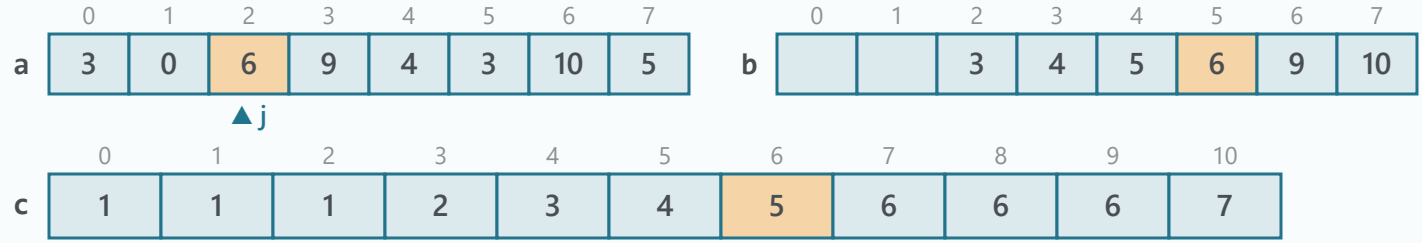
VARIABLES

`j = 3` `a[j] = 9`

`c = [1, 1, 1, 2, 3, 4, 6, 6, 6, 6, 7]`

`b = [_, _, 3, 4, 5, _, 9, 10]`

Code Trace (Continued): Placing into b with L4



L4: a[2] = 6, c[6] = 6 → **b[5] = 6**; c[6]-- = 5.

Step 7 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

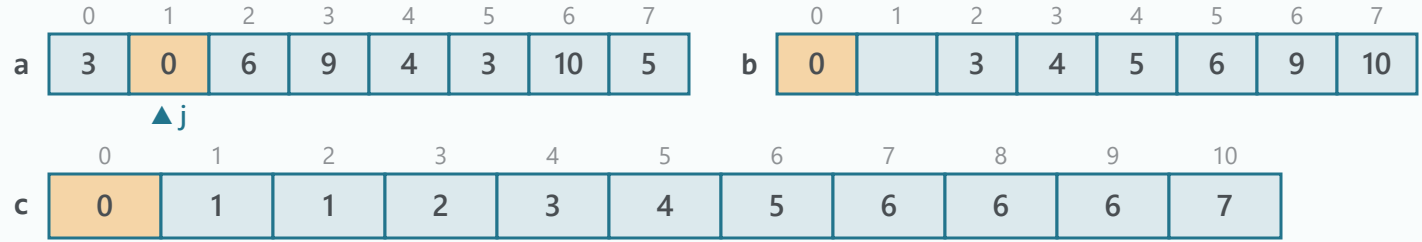
VARIABLES

`j = 2` `a[j] = 6`

`c = [1, 1, 1, 2, 3, 4, 5, 6, 6, 6, 7]`

`b = [_, _, 3, 4, 5, 6, 9, 10]`

Code Trace (Continued): Placing into b with L4



L4: a[1] = 0, c[0] = 1 → **b[0] = 0**; c[0]-- = 0.

Step 8 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

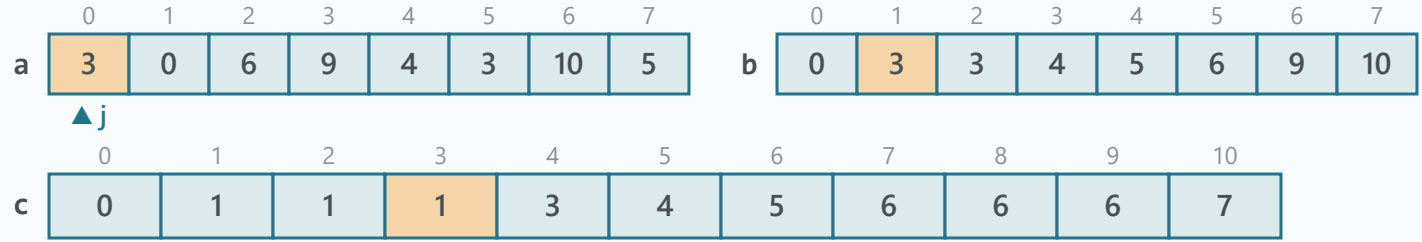
VARIABLES

`j = 1` `a[j] = 0`

`c = [0, 1, 1, 2, 3, 4, 5, 6, 6, 6, 7]`

`b = [0, _, 3, 4, 5, 6, 9, 10]`

Code Trace (Continued): Placing into b with L4



L4: a[0] = 3, c[3] = 2 → **b[1] = 3**; c[3]-- = 1 (the next equal value lands one to the left).

Step 9 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
▶        b[c[a[j]] - 1] = a[j];
▶        c[a[j]]--;
4    }
```

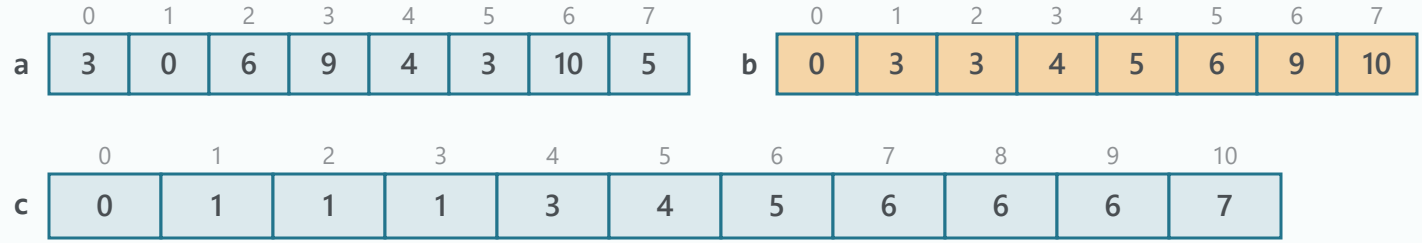
VARIABLES

`j = 0` `a[j] = 3`

`c = [0, 1, 1, 1, 3, 4, 5, 6, 6, 6, 7]`

`b = [0, 3, 3, 4, 5, 6, 9, 10]`

Code Trace (Continued): Placing into b with L4



Result b = [0, 3, 3, 4, 5, 6, 9, 10]. Scanning right to left kept the two 3s in their relative order: **stable**.

Step 10 / 10

```
1    for (int j = r; j >= 0; j--) {          // L4
2        b[c[a[j]] - 1] = a[j];
3        c[a[j]]--;
▶    }
```

VARIABLES

c = [0, 1, 1, 1, 3, 4, 5, 6, 6, 6, 7]

b = [0, 3, 3, 4, 5, 6, 9, 10]

Counting Sort: Analysis and Limits

loop	work	cost
L1	clear c	O(k)
L2	count	O(n)
L3	prefix sums	O(k)
L4	place	O(n)

- Total **O(n + k)**; **O(n)** if k = O(n). Extra memory O(n + k): **not in place**.
- L4 runs right to left, so it is **stable**, a property radix sort requires.

INSTRUCTOR "To sort in linear time we must know the range of the data. If we sort 10 values whose range is 0 to 1000, it becomes an n^3 algorithm... So the range has to be very small."

INSTRUCTOR "In the first pass we count frequencies, in the second we sort. Remember we said the fast transpose works with a single extra array? This is the same trick..."

Code Trace: `shell_sort(a, 8)`

h = 4

0	1	2	3	4	5	6	7
35	33	42	10	14	19	27	44

Gap **h = 4**: elements 4 apart are insertion-sorted separately.

Step 1 / 20

```
1  void shell_sort(int a[], int n) {
▶       for (int h = n / 2; h > 0; h /= 2) // gap
3           for (int i = h; i < n; i++) {
4               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // shift back h
7                   j -= h;
8               }
9               a[j] = v;
10          }
11 }
```

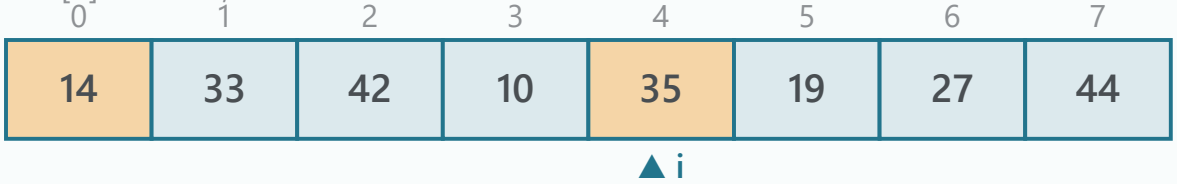
VARIABLES

h = 4

a = [35, 33, 42, 10, 14, 19, 27, 44]

Code Trace: `shell_sort(a, 8)`

h = 4 · a[4] chain: 0, 4



i = 4: v = 14; shifted back by 4 once, a[0] = 14.

Step 2 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // shift back h
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

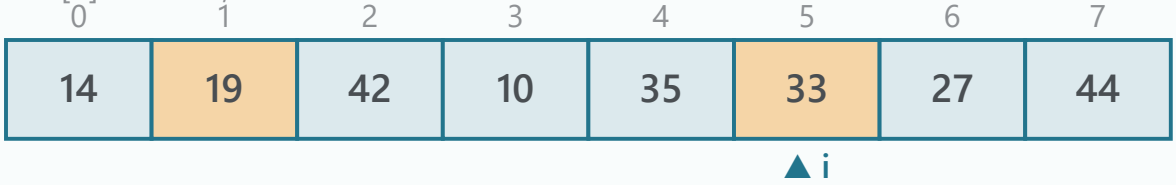
VARIABLES

h = 4 | i = 4 | v = 14 | j = 0

a = [14, 33, 42, 10, 35, 19, 27, 44]

Code Trace: `shell_sort(a, 8)`

h = 4 · a[5] chain: 1, 5



i = 5: v = 19; shifted back by 4 once, a[1] = 19.

Step 3 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // shift back h
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

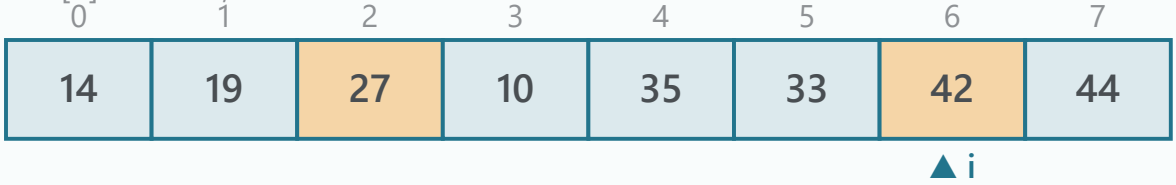
VARIABLES

h = 4 | i = 5 | v = 19 | j = 1

a = [14, 19, 42, 10, 35, 33, 27, 44]

Code Trace: `shell_sort(a, 8)`

h = 4 · a[6] chain: 2, 6



i = 6: v = 27; shifted back by 4 once, a[2] = 27.

Step 4 / 20

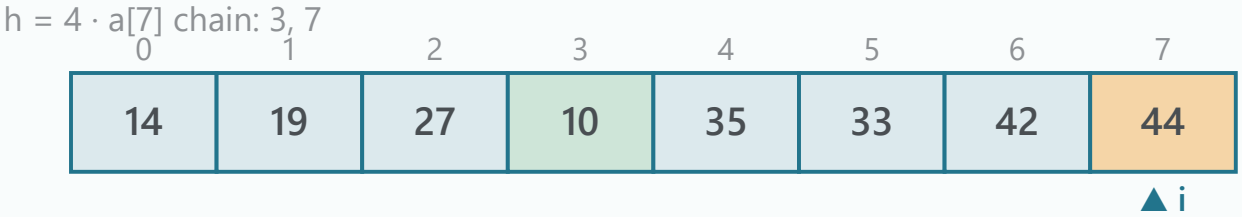
```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // shift back h
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

VARIABLES

h = 4 | i = 6 | v = 27 | j = 2

a = [14, 19, 27, 10, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`



i = 7: v = 44; a[3] ≤ 44, stays in place.

Step 5 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // shift back h
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

VARIABLES

h = 4 | i = 7 | v = 44 | j = 7

a = [14, 19, 27, 10, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 2

0	1	2	3	4	5	6	7
14	19	27	10	35	33	42	44

Gap **h = 2**: elements 2 apart are insertion-sorted separately.

Step 6 / 20

```
1  void shell_sort(int a[], int n) {
▶      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
5              while (j >= h && a[j - h] > v) {
6                  a[j] = a[j - h];    // shift back h
7                  j -= h;
8              }
9              a[j] = v;
10         }
11 }
```

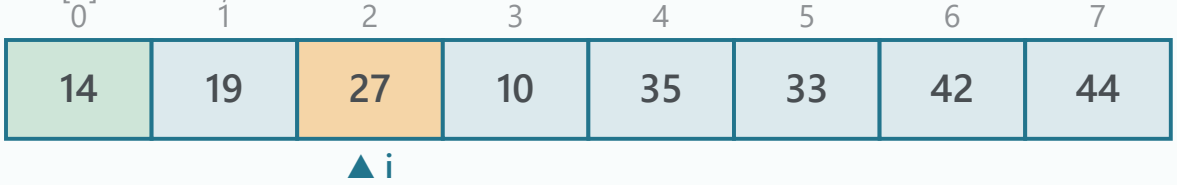
VARIABLES

h = 2

a = [14, 19, 27, 10, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 2 · a[2] chain: 0, 2



i = 2: v = 27; a[0] ≤ 27, stays in place.

Step 7 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // shift back h
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

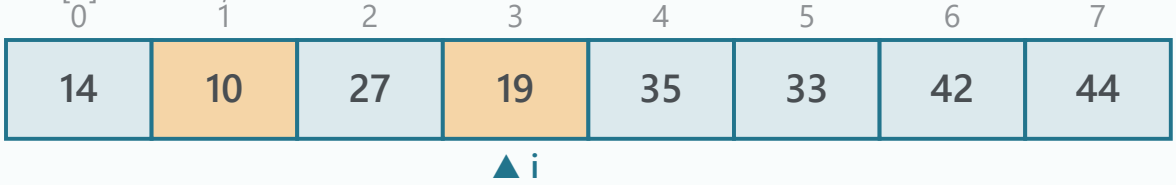
VARIABLES

h = 2 | i = 2 | v = 27 | j = 2

a = [14, 19, 27, 10, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 2 · a[3] chain: 1, 3



i = 3: v = 10; shifted back by 2 once, a[1] = 10.

Step 8 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // shift back h
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

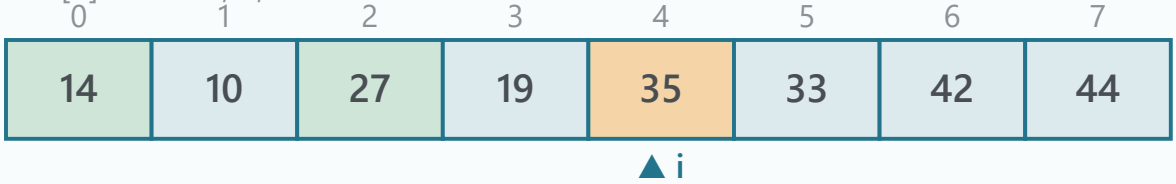
VARIABLES

h = 2 | i = 3 | v = 10 | j = 1

a = [14, 10, 27, 19, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 2 · a[4] chain: 0, 2, 4



i = 4: v = 35; a[2] ≤ 35, stays in place.

Step 9 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // shift back h
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

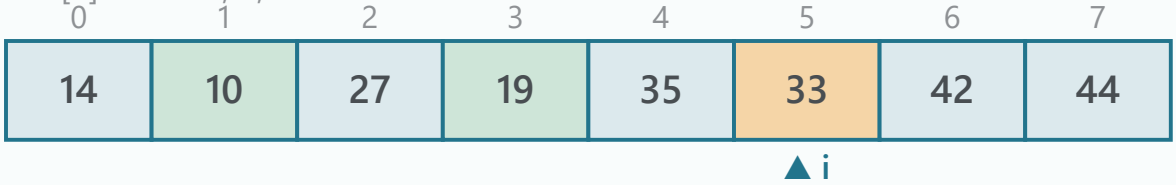
VARIABLES

h = 2 | i = 4 | v = 35 | j = 4

a = [14, 10, 27, 19, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 2 · a[5] chain: 1, 3, 5



i = 5: v = 33; a[3] ≤ 33, stays in place.

Step 10 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // shift back h
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

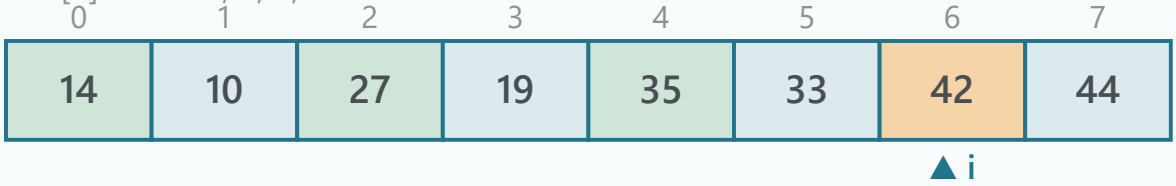
VARIABLES

h = 2 | i = 5 | v = 33 | j = 5

a = [14, 10, 27, 19, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 2 · a[6] chain: 0, 2, 4, 6



i = 6: v = 42; a[4] ≤ 42, stays in place.

Step 11 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
▶             int v = a[i], j = i;
5             while (j >= h && a[j - h] > v) {
6                 a[j] = a[j - h];    // shift back h
7                 j -= h;
8             }
▶             a[j] = v;
10        }
11 }
```

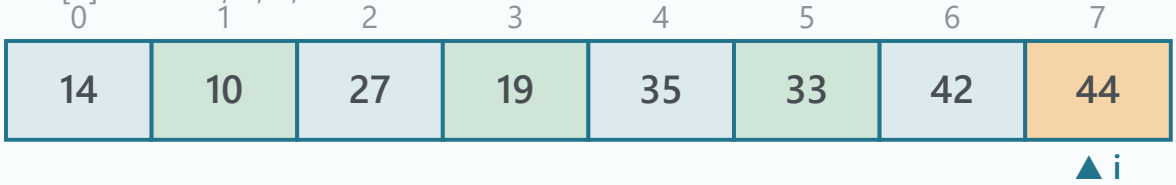
VARIABLES

h = 2 | i = 6 | v = 42 | j = 6

a = [14, 10, 27, 19, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 2 · a[7] chain: 1, 3, 5, 7



i = 7: v = 44; a[5] ≤ 44, stays in place.

Step 12 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // shift back h
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

VARIABLES

h = 2 | i = 7 | v = 44 | j = 7

a = [14, 10, 27, 19, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 1

0	1	2	3	4	5	6	7
14	10	27	19	35	33	42	44

Last pass **h = 1**: plain insertion sort, but the array is now nearly sorted.

Step 13 / 20

```
1  void shell_sort(int a[], int n) {
▶      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
5              while (j >= h && a[j - h] > v) {
6                  a[j] = a[j - h];    // shift back h
7                  j -= h;
8              }
9              a[j] = v;
10         }
11 }
```

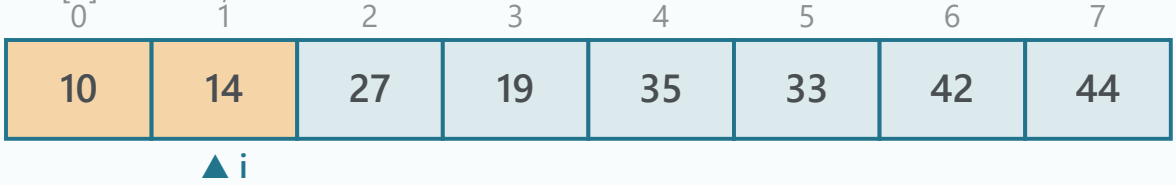
VARIABLES

h = 1

a = [14, 10, 27, 19, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 1 · a[1] chain: 0, 1



i = 1: v = 10; shifted back by 1 once, a[0] = 10.

Step 14 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // shift back h
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

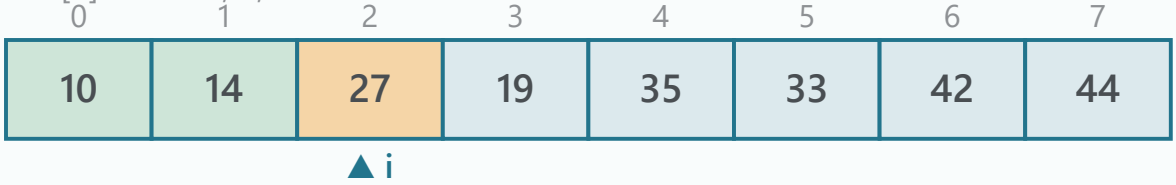
VARIABLES

h = 1 | i = 1 | v = 10 | j = 0

a = [10, 14, 27, 19, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 1 · a[2] chain: 0, 1, 2



i = 2: v = 27; a[1] ≤ 27, stays in place.

Step 15 / 20

```
1 void shell_sort(int a[], int n) {
2     for (int h = n / 2; h > 0; h /= 2) // gap
3         for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // shift back h
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11  }
```

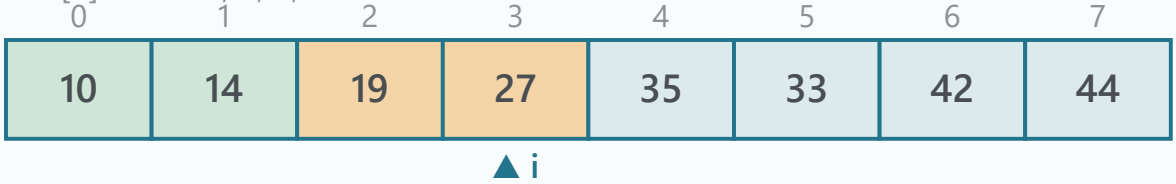
VARIABLES

h = 1 | i = 2 | v = 27 | j = 2

a = [10, 14, 27, 19, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 1 · a[3] chain: 0, 1, 2, 3



i = 3: v = 19; shifted back by 1 once, a[2] = 19.

Step 16 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // shift back h
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

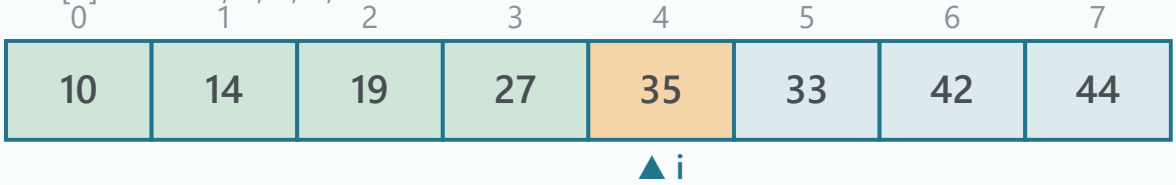
VARIABLES

h = 1 | i = 3 | v = 19 | j = 2

a = [10, 14, 19, 27, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 1 · a[4] chain: 0, 1, 2, 3, 4



i = 4: v = 35; a[3] ≤ 35, stays in place.

Step 17 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // shift back h
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

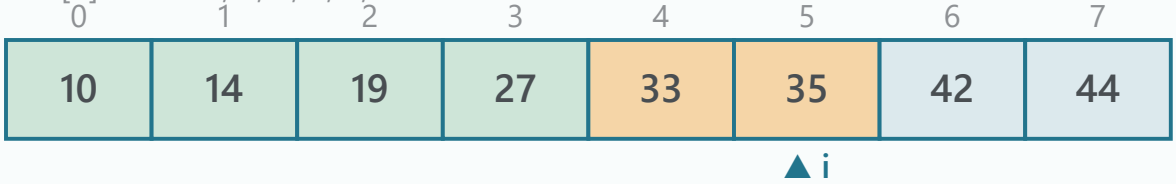
VARIABLES

h = 1 | i = 4 | v = 35 | j = 4

a = [10, 14, 19, 27, 35, 33, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 1 · a[5] chain: 0, 1, 2, 3, 4, 5



i = 5: v = 33; shifted back by 1 once, a[4] = 33.

Step 18 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
4              int v = a[i], j = i;
▶              while (j >= h && a[j - h] > v) {
▶                  a[j] = a[j - h];    // shift back h
7                  j -= h;
8              }
▶              a[j] = v;
10         }
11 }
```

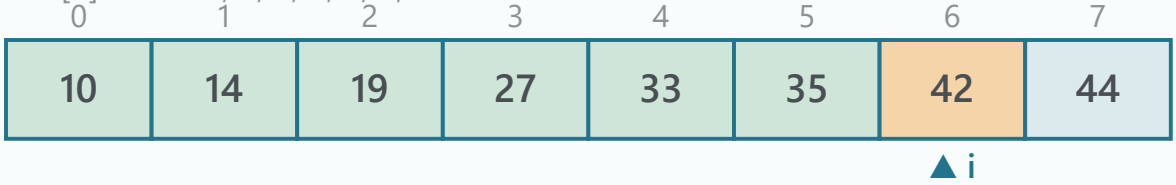
VARIABLES

h = 1 | i = 5 | v = 33 | j = 4

a = [10, 14, 19, 27, 33, 35, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 1 · a[6] chain: 0, 1, 2, 3, 4, 5, 6



i = 6: v = 42; a[5] ≤ 42, stays in place.

Step 19 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // shift back h
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

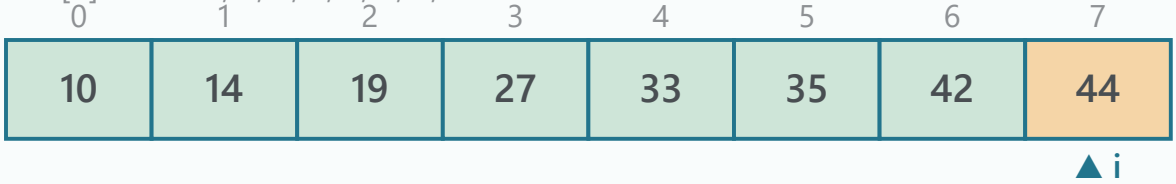
VARIABLES

h = 1 | i = 6 | v = 42 | j = 6

a = [10, 14, 19, 27, 33, 35, 42, 44]

Code Trace: `shell_sort(a, 8)`

h = 1 · a[7] chain: 0, 1, 2, 3, 4, 5, 6, 7



i = 7: v = 44; a[6] ≤ 44, stays in place.

Step 20 / 20

```
1  void shell_sort(int a[], int n) {
2      for (int h = n / 2; h > 0; h /= 2) // gap
3          for (int i = h; i < n; i++) {
▶               int v = a[i], j = i;
5               while (j >= h && a[j - h] > v) {
6                   a[j] = a[j - h];    // shift back h
7                   j -= h;
8               }
▶               a[j] = v;
10          }
11 }
```

VARIABLES

h = 1 | i = 7 | v = 44 | j = 7

a = [10, 14, 19, 27, 33, 35, 42, 44]

Shell Sort (1959)

- The weakness of insertion sort: an element moves only **one** position at a time; $O(n^2)$ on reverse-sorted input.
- Shell: first sort **distant** elements (gap h), then shrink h; the last pass uses h = 1, so the result is **always correct**.
- The cost depends on the gap sequence: worst case $O(n^2)$ for h = n/2, n/4, … (Shell's); $O(n^{3/2})$ for Knuth's 1, 4, 13, 40, …
- In place, but long-distance moves make it **not stable**.

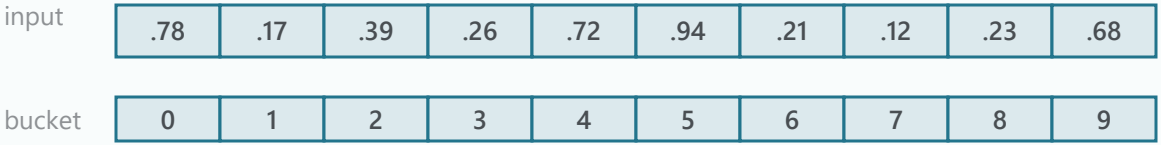
Combined use

When properties of the data are known, Shell sort and bucket sort can be used together.

In practice

Short code and good practical speed: a favorite in embedded systems and for small to medium arrays.

Bucket Sort

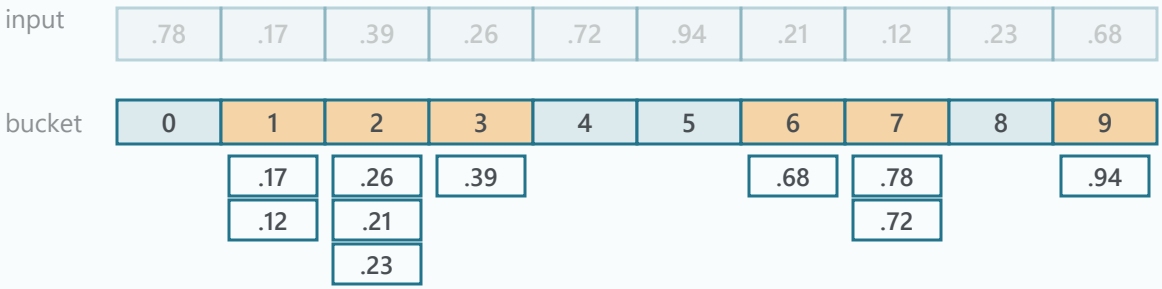


Input: 10 numbers in [0, 1); 10 buckets, each 0.1 wide.

Step 1 / 4

If the input is **uniformly distributed** over the range, each bucket holds O(1) elements on average → O(n) on average (CLRS). If everything falls into one bucket, the worst case is the cost of sorting within the bucket ($O(n^2)$ with insertion sort). Extra memory O(n + number of buckets).

Bucket Sort

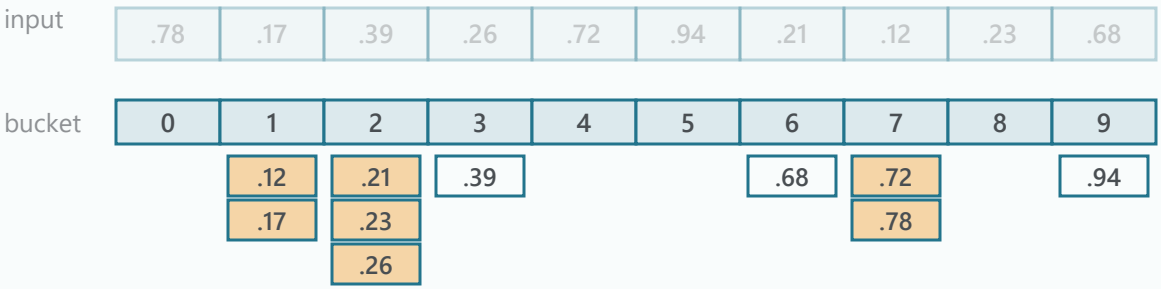


Distribute: value x goes into bucket ⌊10·x⌋ (e.g. .78 → 7, .17 → 1). O(n).

Step 2 / 4

If the input is **uniformly distributed** over the range, each bucket holds O(1) elements on average → O(n) on average (CLRS). If everything falls into one bucket, the worst case is the cost of sorting within the bucket ($O(n^2)$ with insertion sort). Extra memory O(n + number of buckets).

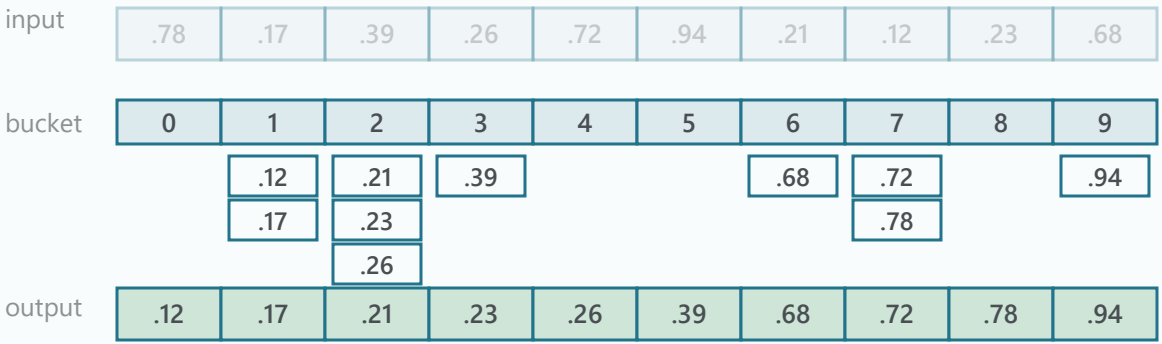
Bucket Sort



Each bucket is sorted on its own (by insertion sort): cheap, since buckets are small.

If the input is **uniformly distributed** over the range, each bucket holds O(1) elements on average → O(n) on average (CLRS). If everything falls into one bucket, the worst case is the cost of sorting within the bucket ($O(n^2)$ with insertion sort). Extra memory O(n + number of buckets).

Bucket Sort



Concatenate the buckets in order → sorted output. **O(n)** on average for a uniform distribution.

Step 4 / 4

If the input is **uniformly distributed** over the range, each bucket holds O(1) elements on average → O(n) on average (CLRS). If everything falls into one bucket, the worst case is the cost of sorting within the bucket ($O(n^2)$ with insertion sort). Extra memory O(n + number of buckets).

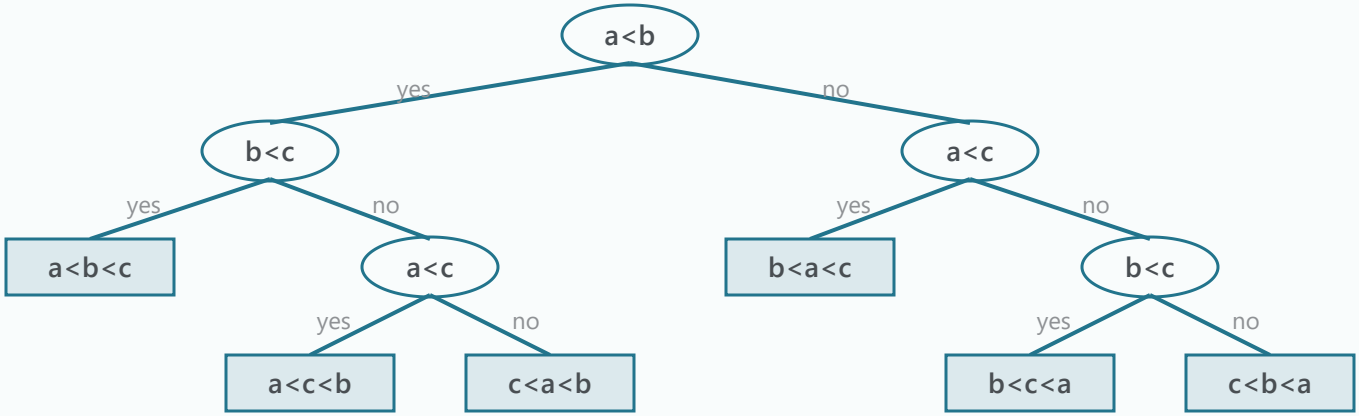
Bucket Sort: An Implementation in C

```c
typedef struct bnode { double x; struct bnode *next; } bnode;

int bucket_sort(double a[], int n) {        /* 0 <= a[i] < 1 */
    bnode **bucket = calloc(n, sizeof *bucket);
    bnode *pool = malloc(n * sizeof *pool);
    if (!bucket || !pool) { free(bucket); free(pool); return -1; }
    for (int i = 0; i < n; i++) {           /* scatter + sorted insert */
        bnode **p = &bucket[(int)(a[i] * n)];
        while (*p && (*p)->x <= a[i]) p = &(*p)->next;
        pool[i].x = a[i]; pool[i].next = *p; *p = &pool[i];
    }
    int k = 0;                              /* gather the buckets */
    for (int b = 0; b < n; b++)
        for (bnode *q = bucket[b]; q; q = q->next) a[k++] = q->x;
    free(bucket); free(pool);
    return 0;
}
```

- n buckets; the value `a[i]` is inserted **in order** into linked list `⌊a[i]·n⌋` (insertion sort within the bucket).
- Thanks to `≤`, equal values go to the end → stable. The nodes come from a single `pool` array: two allocations instead of n separate `malloc`s, and one pair of `free`s.

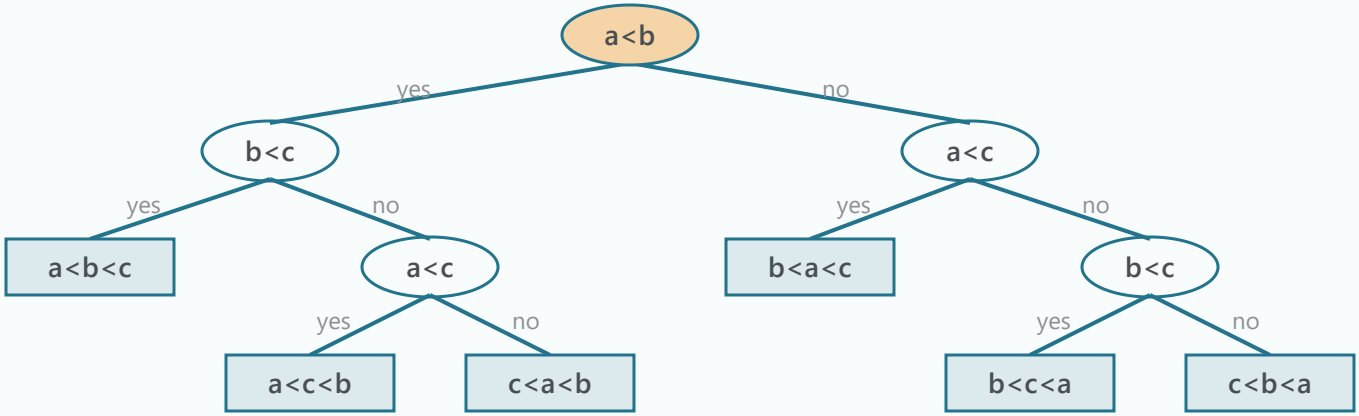
Optimal Sorting Time: The Decision Tree



Tests are ellipses, decisions (orderings) rectangles. For n = 3 there are 3! = 6 leaves.

Every comparison sort is a decision tree: each internal node is a `less()` call, each leaf an ordering of the input. The worst-case number of comparisons = the tree's **height**.

Optimal Sorting Time: The Decision Tree

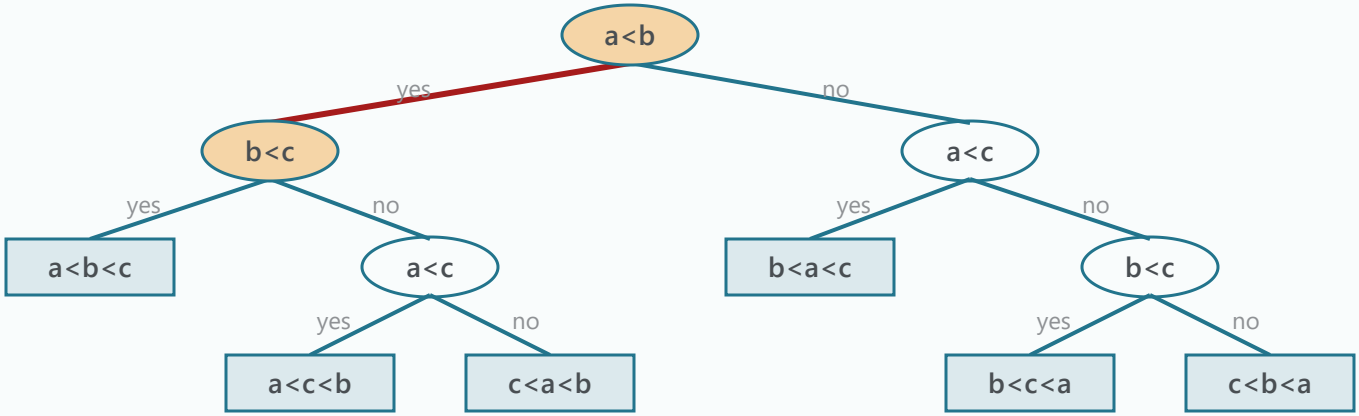


Input a = 5, b = 8, c = 2. Root: a < b? 5 < 8 → **yes**.

Step 2 / 5

Every comparison sort is a decision tree: each internal node is a `less()` call, each leaf an ordering of the input. The worst-case number of comparisons = the tree's **height**.

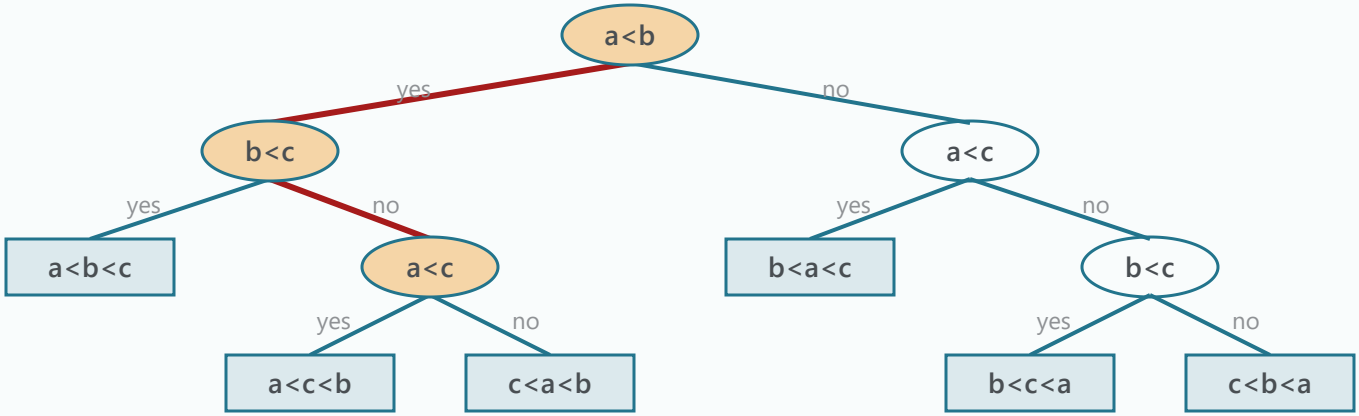
Optimal Sorting Time: The Decision Tree



b < c? 8 < 2 → **no**: c must come before b.

Every comparison sort is a decision tree: each internal node is a `less()` call, each leaf an ordering of the input. The worst-case number of comparisons = the tree's **height**.

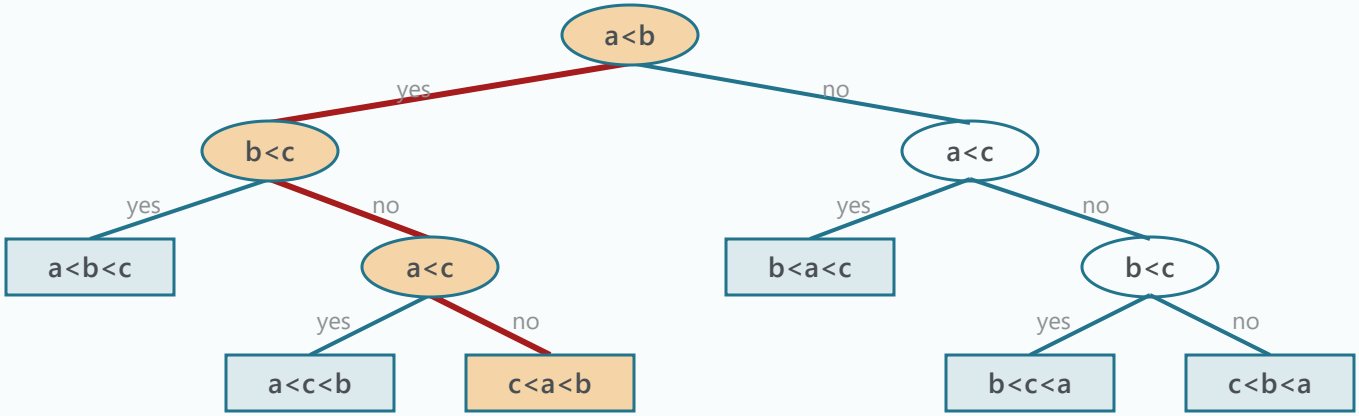
Optimal Sorting Time: The Decision Tree



a < c? 5 < 2 → **no**.

Every comparison sort is a decision tree: each internal node is a `less()` call, each leaf an ordering of the input. The worst-case number of comparisons = the tree's **height**.

Optimal Sorting Time: The Decision Tree



Leaf **c < a < b** → 2, 5, 8. For this input, 3 comparisons = the length of the path.

Every comparison sort is a decision tree: each internal node is a `less()` call, each leaf an ordering of the input. The worst-case number of comparisons = the tree's **height**.

Theorem: The Lower Bound for Comparison Sorting

Theorem. Every decision tree that sorts n distinct elements has depth (number of levels) at least $\lceil \log_2(n!) \rceil + 1$; so in the worst case at least $\lceil \mathbf{lg(n!)} \rceil$ **comparisons** are needed.

- There are at least n! leaves (one per ordering).
- A binary tree of height h has at most 2^h leaves: $2^h \geq n! \Rightarrow h \geq \lg(n!)$.
- Stirling: $\lceil \lg(n!) \rceil = n \lg n - n \lg e + \Theta(\lg n) \approx$ **$n \lg n - 1.4427\, n$**.
- Result: comparison sorting is **$\Omega(n \lg n)$**; counting, bucket and radix sort make no comparisons, so the bound does not apply to them.

n	$\lceil \lg n! \rceil$	$n \lg n - 1.44n$	$n(n-1)/2$
3	3	0.4	3
5	7	4.4	10
10	22	18.8	45
20	62	57.6	190
100	525	520.1	4950

INSTRUCTOR “We now have a touchstone for comparing the algorithms we will write from here on.”

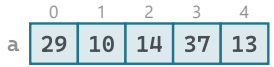
Lab: Basic Sorts

Selection: 29, 10, 14, 37, 13

Input (comma-separated numbers) 29, 10, 14, 37, 13 Apply

CODE: selection

```
1 void selection(int a[], int L, int R) {
2     for (int i = L; i < R; i++) {
3         int min = i;
4         for (int j = i + 1; j <= R; j++)
5             if (less(a[j], a[min]))
6                 min = j;
7         SWAP(a[i], a[min]);
8     }
9 }
```



comparisons 0

OUTPUT (none yet)

Input: [29, 10, 14, 37, 13]. Selection sort: in each round the smallest item of the remaining part is found and put where the round started.

SUMMARY

Counts on the Same Input (n = 1000)

algorithm	sorted: comp. / moves	reversed: comp. / moves	random: comp. / moves
Selection	499,500 / 999	499,500 / 999	499,500 / 999
Insertion	999 / 0	499,500 / 499,500	247,876 / 246,884
Bubble (early exit)	999 / 0	499,500 / 499,500	499,035 / 246,884
Shell (n/2, n/4, ...)	8,006 / 0	11,716 / 4,700	14,760 / 7,254

Counted by a Python simulation (random input: seed 19). Moves: swaps for selection and bubble, shifts for insertion and Shell.

Selection

Blind to the input: 499,500 comparisons and 999 swaps every time.

Insertion

999 comparisons when sorted; ≈ $n^2/4$ on random input; $n^2/2$ when reversed.

Shell

Far less work than insertion on random input: long shifts remove inversions fast.

SUMMARY

Complexity Summary

algorithm	best	average	worst	extra memory	stable	in place
Selection	$n^2/2$	$n^2/2$	$n^2/2$	O(1)	no	yes
Insertion	O(n)	$\approx n^2/4$	$n^2/2$	O(1)	yes	yes
Bubble (early exit)	O(n)	$\approx n^2/2$ comp.	$n^2/2$	O(1)	yes	yes
Shell (Knuth h)	O(n lg n)	—	$O(n^{3/2})$	O(1)	no	yes
Counting	O(n+k)	O(n+k)	O(n+k)	O(n+k)	yes	no
Bucket (uniform)	O(n)	O(n)	$O(n^2)$	O(n)	yes*	no

For selection, insertion and bubble the figures are comparison counts. * If a stable sort is used inside each bucket. Shell's average depends on the gap sequence and has no known closed form. The lower bound for comparison sorting is $\lceil \lg n! \rceil \approx n \lg n$.

SUMMARY

Which One When?

Swaps/writes are expensive

Selection: exactly $n - 1$ swaps.

Small or nearly sorted

Insertion: stable, in place, $O(n)$ when nearly sorted; also used on the small pieces in quicksort.

Teaching / sortedness test

Bubble with early exit: detects a sorted array in one pass; rarely used in production.

Small integer range

Counting: $O(n + k)$, stable; if $k \gg n$, memory and time are wasted.

Medium size, no extra memory

Shell: a few lines of code, clearly faster than $O(n^2)$.

Uniform on [0,1)

Bucket: $O(n)$ on average.

SUMMARY

Common Mistakes

1. Keeping the minimum **value** in selection sort: the swap needs its **index** (`min`).
2. Not saving `v = a[i]` before shifting in insertion sort: the first shift overwrites a[i].
3. Writing `j ≥ 0`: for a subarray a[L..R] the bound is `j ≥ L`.
4. Getting the direction wrong in `if (less(A, B)) SWAP(A, B)`: it swaps when A < B.
5. Ignoring k in counting sort: a negative value or one above k writes outside `c[]`.
6. Running L4 of counting sort left to right: it still sorts, but **stability** is lost.

Exercises

1. Write the array after each pass of selection sort on [5, 1, 4, 2, 8]. How many swaps were made?
2. How many comparisons does insertion sort make on [1, 2, 3, …, n] and on [n, …, 2, 1]?
3. How many passes does `bubble2` run on [1, 2, 3, 5, 4]?
4. For a = [2, 5, 3, 0, 2, 3, 0, 3], k = 5, write the array c after L3 of counting sort.
5. What is the minimum height of a decision tree for 3 elements? For 4?
6. Find the smallest example showing that selection sort is not stable.

▶ **Show answers**

SUMMARY

Chapter Summary

Three $O(n^2)$ algorithms

Same class, different character: selection makes few swaps, insertion is fast on nearly sorted data, bubble can stop early.

Average case

The expected value over all n! orderings: $\approx n^2/4$ comparisons for insertion. Not the mean of best and worst!

Non-comparison

Counting O(n+k) and bucket O(n) on average: knowledge of the data (range, distribution) buys speed.

Concepts

Stability (order of equal keys) and in-place operation (extra memory) drive the choice of algorithm.

Shell

Insertion sort with gaps; the final pass with h = 1 guarantees correctness.

Lower bound

Comparison sorting cannot beat $\lceil \lg n! \rceil \approx n \lg n$ comparisons → the touchstone for advanced sorting algorithms.

SUMMARY

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 7 (Sorting).
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapter 6 (Elementary Sorting Methods): the macro approach, Shell sort.
- T. H. Cormen et al., *Introduction to Algorithms* (CLRS), Chapter 2 (insertion), 8 (lower bound, counting, bucket).

← Chapter 18: Shortest Paths and Activity Networks

Course page

Chapter 20: Advanced Sorting Algorithms

The End

Chapter 19: Basic Sorting Algorithms

Next chapter: Chapter 20: Advanced Sorting Algorithms · Course page