

Recursive insertion sort: a sorted prefix

Task: sort the first n elements of a in ascending order.

1. **Base case:** $n \leq 1$. An empty or one-element array is already sorted.
2. **Smaller problem:** recursively sort the first $n - 1$ elements.
3. **Work after the call:** insert $a[n - 1]$ into that sorted prefix.

$$[5, 2, 4, 1] \longrightarrow \left[\underbrace{2, 4, 5}_{\text{sorted prefix}}, 1 \right] \longrightarrow [1, 2, 4, 5].$$

The recursive call changes the array. This function returns `void`.

Recursive insertion sort in C

```
void insertion_sort(int a[], int n)
{
    if (n <= 1) return;
    insertion_sort(a, n - 1);
    int key = a[n - 1];
    int j = n - 2;
    while (j >= 0 && a[j] > key) {
        a[j + 1] = a[j];
        j--;
    }
    a[j + 1] = key;
}
```

After the recursive call, $a[0]$ through $a[n - 2]$ are sorted.

The loop shifts larger elements right. $j \geq 0$ prevents reading $a[-1]$.

Insertion sort: work happens as calls return

For [5, 2, 4, 1], the calls use $n = 4, 3, 2, 1$. The $n = 1$ call returns immediately.

Call resumes	Key	Existing elements shifted right	Array afterward
$n = 2$	2	5	[2, 5, 4, 1]
$n = 3$	4	5	[2, 4, 5, 1]
$n = 4$	1	5, then 4, then 2	[1, 2, 4, 5]

Why do we wait for the recursive call?

The insertion step needs a sorted prefix. That prefix is ready only after `insertion_sort(a, n - 1)` finishes.

Quiz 9

Run recursive insertion sort on $[4, 3, 2, 1]$.

1. Write the array after the calls with $n = 2$ and $n = 3$ finish.
2. How many existing elements shift right when $n = 4$ resumes?
3. How many such shifts occur over the entire sort?

Quiz 9

Run recursive insertion sort on [4, 3, 2, 1].

1. Write the array after the calls with $n = 2$ and $n = 3$ finish.
2. How many existing elements shift right when $n = 4$ resumes?
3. How many such shifts occur over the entire sort?

Answer

After $n = 2$: [3, 4, 2, 1]. After $n = 3$: [2, 3, 4, 1].

The last insertion shifts three elements. Total shifts: $1 + 2 + 3 = 6$.

For a decreasing array of length n , the total is $n(n - 1)/2$.

Merge sort: sort both halves, then combine

Base case: an interval with at most one element is already sorted.

Stage	Example: sorting [7, 2, 6, 3]
Split	Left half [7, 2] and right half [6, 3]
Sort left half	Split into [7] and [2], then merge to get [2, 7]
Sort right half	Split into [6] and [3], then merge to get [3, 6]
Merge both halves	Combine [2, 7] and [3, 6] to get [2, 3, 6, 7]

Each call waits for **both halves** to be sorted before it merges them.

Merging two sorted halves

Compare the first unused element of each half. Copy the smaller one to `tmp`.

Left remaining	Right remaining	Action	Output so far
[2, 7]	[3, 6]	Copy 2 from the left	[2]
[7]	[3, 6]	Copy 3 from the right	[2, 3]
[7]	[6]	Copy 6 from the right	[2, 3, 6]
[7]	Empty	Copy the remaining 7	[2, 3, 6, 7]

The smallest unused element must be at the front of one of the sorted halves.

Finally, copy the merged interval from `tmp` back into `a`.

Merge sort: the recursive function

```
void merge_sort(int a[], int tmp[], int lo, int hi)
{
    if (lo >= hi) return;
    int mid = lo + (hi - lo) / 2;
    merge_sort(a, tmp, lo, mid);
    merge_sort(a, tmp, mid + 1, hi);
    merge(a, tmp, lo, mid, hi);
}
```

lo and hi are inclusive indices. Define merge before this function.

For four elements, declare int tmp[4]; and call merge_sort(a, tmp, 0, 3).

The same temporary array is shared by all calls, each using its own interval.

The merge helper: choosing the next element

```
void merge(int a[], int tmp[], int lo, int mid, int hi)
{
    int i = lo, j = mid + 1, k = lo;
    while (i <= mid && j <= hi) {
        if (a[i] <= a[j]) tmp[k++] = a[i++];
        else tmp[k++] = a[j++];
    }
    /* Remaining steps on the next slide. */
}
```

- i scans the left half; j scans the right half.
- k is the next free position in tmp.
- tmp[k++] = a[i++] copies one value, then advances both indices.

The merge helper: leftovers and copy-back

```
/* Continue inside merge, after the comparison loop. */
while (i <= mid) tmp[k++] = a[i++];
while (j <= hi) tmp[k++] = a[j++];
for (int p = lo; p <= hi; p++) a[p] = tmp[p];
}
```

- When one half is exhausted, the other half may still have elements.
- Copy those remaining elements in their current order.
- Copy back only positions `lo` through `hi`.

Merging an interval of m elements takes $O(m)$ time.

Every element is copied once into `tmp` and once back into `a`.