CS617 | Chapter 02 | NOTES

Sorting Algorithms and Experimental Evidence

Alex Barragan and Sol | Xandrian Labs

Harmonized teaching edition | Release snapshot | 4 October 2026

Contents

CS617 Chapter 02 **1**
Sorting Algorithms: Bubble, Shell, and the Evidence Between Them . . . 1
Purpose . . . 1
1. Data model . . . 1
2. Ordering . . . 1
3. Data capacity . . . 2
4. Deterministic input patterns . . . 2
5. Uniqueness constraint . . . 3
6. Sorting algorithms . . . 3
7. Comparison instrumentation . . . 3
8. Measurement boundary . . . 4
9. Setup and function keys . . . 4
10. Post-sort assertion . . . 5
11. Successful run report . . . 6
12. Explicit non-goals . . . 6
13. Acceptance criteria . . . 6
14. The boundary between WHAT and HOW . . . 6
A final thought . . . 7

CS617 Chapter 02

Sorting Algorithms: Bubble, Shell, and the Evidence Between Them

Student edition — harmonized teaching reconstruction, release snapshot (4 October 2026). Derived from the frozen Chapter 02 `what.md`; it is not a recovered original handout and does not add grading requirements to the historical assignment. Target: Casio PB-2000C.

Purpose

Sorting a collection seems simple: put its keys in order. Yet two correct algorithms can perform very different amounts of work, and even a program that *looks* sorted can tell us remarkably little about what happened inside the processor.

Build one standalone laboratory with **optimized Bubble sort** and **halving-gap Shell sort**. Give both algorithms the **same reproducible input**, measure comparisons at two different levels, and check their output without contaminating the measurements. The experiment is successful only when another engineer can understand and reproduce what you observed.

> Control the input. Define the unit of work. Measure the sorting run. Check the result independently of the measurement.

Your task is the observable **WHAT**, not a prescribed register map, comparator subroutine layout, memory address, implementation sequence or historical reference solution. The constraints below are binding; the specific HOW remains yours.

1. Data model

The sortable data is an array of `n` fixed-width keys.

Each key is exactly `m` bytes long.

```
record 0:  m bytes
record 1:  m bytes
...
record n-1: m bytes
```

The sort operates **in situ**: the original array is rearranged in place.

A temporary key of `m` bytes and other fixed-size working state are allowed. A second array of `n * m` bytes is not.

2. Ordering

Keys are sorted in **ascending binary lexicographic order**.

Each byte is interpreted as an unsigned value in the range `0..255`.

Comparison proceeds from the first byte of the key toward the last byte. At the first unequal byte, the relation `<` or `>` is known and comparison stops. Keys are equal only if all `m` bytes are equal.

No character-set, string, numeric, sign, case, terminator, or locale semantics are applied to the key.

3. Data capacity

The data array has a compile-time capacity named conceptually as:

`DATA_BYTES`

For this assignment:

`DATA_BYTES = 8192`

The array reservation is governed by this one constant so that a future version can raise the capacity to 16 KiB or more without changing the conceptual design of the assignment.

A setup is valid only when:

```
n >= 1
m >= 1
n * m <= DATA_BYTES
```

`n = 0` and `m = 0` are explicitly invalid configurations.

The 8 KiB limit applies to the data array itself. Temporary storage and other program state are expected to have a smaller constant footprint outside that allowance.

4. Deterministic input patterns

The program shall provide three initialization patterns.

ORDERED

Generate unique keys representing:

`1, 2, 3, ... n`

The integer value is represented in exactly `m` bytes, most-significant byte first, so that binary lexicographic order and the generated numeric order agree.

Example for `m = 2`:

```
00 01
00 02
```

```
00 03
...
00 FF
01 00
01 01
...
```

REVERSE

Generate exactly the same set of keys as `ORDERED`, but place them in reverse order.

For example, if the ordered set is:

```
00 01
00 02
00 03
00 04
00 05
```

the reverse input is:

```
00 05
00 04
00 03
00 02
00 01
```

`ORDERED` and `REVERSE` therefore differ only in permutation, not in the population of keys being sorted.

SAME

Every key is:

`FF FF ... FF`

for all `m` bytes.

This intentionally forces equal-key comparisons to inspect the entire key.

5. Uniqueness constraint

`ORDERED` and `REVERSE` must contain unique keys.

Because the generated sequence begins at 1, the configuration must satisfy:

`n <= 256^m - 1`

For a one-byte key this means:

`m = 1  =>  n <= 255`

A configuration that cannot satisfy the uniqueness requirement must be rejected during setup.

`SAME` is intentionally a multiset and is exempt from the uniqueness requirement.

6. Sorting algorithms

Bubble sort

Implement an optimized in-place Bubble sort.

Required behavior:

- compare adjacent keys;
- swap only when the left key is greater than the right key;
- shrink the unsorted region after each pass;
- terminate early when an entire pass performs no swaps.

Stability is not a requirement of this assignment.

Shell sort

Implement an in-place Shell sort using gapped insertion sorting.

Use the simple halving gap sequence:

```
gap = n / 2
gap = gap / 2
...
gap = 1
```

The sequence must terminate with `gap = 1`.

Whole `m`-byte records are moved as necessary.

Stability is not a requirement.

7. Comparison instrumentation

Two distinct comparison counters shall be maintained and reported with the following user-facing vocabulary.

7.1 KEY-CMP

`KEY-CMP` counts algorithm-level key comparisons.

Increment it once whenever the sorting algorithm asks for the relation:

```
A[i] ? A[j]
```

A complete comparison of two keys counts as one `KEY-CMP` regardless of how many machine instructions are required to determine the result.

7.2 INST-CMP

`INST-CMP` counts the actual machine comparison/check instructions executed while comparing key data against key data.

One executed key-data comparison/check instruction counts as one `INST-CMP`.

If a machine instruction legitimately compares more than one byte of key data, it still counts as one `INST-CMP`: the metric counts executed instructions, not bytes.

Loads, stores, address calculations, index tests, loop tests, gap tests, setup validation, and other control-flow operations do not count.

Comparison of a key must short-circuit. Once an instruction establishes `<` or `>`, no remaining portion of that key pair is examined.

Equal keys necessarily require comparison through the full key.

7.3 Vocabulary rationale

The labels are deliberately pedagogical:

```
KEY-CMP   algorithm-level work
INST-CMP  processor-instruction work
ORDER     independent post-sort assertion
```

`INST-CMP` is preferred over `BYTE-CMP` because the defined metric counts executed comparison instructions rather than bytes. `ORDER=PASS` / `ORDER=FAIL` is preferred over `SORTED=YES` / `SORTED=NO` because the field reports the result of an independent post-sort assertion rather than merely repeating the sorter's intended outcome.

8. Measurement boundary

The reported counters measure the sorting algorithm only.

Before each sort:

```
regenerate configured input
clear KEY-CMP
```

```
clear INST-CMP
run selected sort
```

The final sortedness assertion is excluded from both counters.

Bubble sort and Shell sort must therefore begin from the same configured input rather than one algorithm receiving the output left by the other.

9. Setup and function keys

FK1 - SETUP

FK1 enters a small setup form for:

```
n
m
input pattern: ORDERED / REVERSE / SAME
```

The setup must reject invalid configurations.

At minimum this includes:

```
n < 1
m < 1
n * m > DATA_BYTES
ORDERED or REVERSE with more unique values requested than m bytes can represent
```

After a valid setup is accepted, the configured deterministic input is generated immediately before returning to the ready state. This allows the current data image to be inspected before either sorting algorithm runs.

The default startup configuration is likewise generated once when the application enters its ready state.

FK2 - BUBBLE

FK2:

```
regenerates the configured input
displays a static BUBBLE / SORTING... status screen
clears the counters
runs optimized Bubble sort
asserts ascending sortedness
displays the report if valid
```

FK3 - SHELL

FK3:

```
regenerates the configured input
displays a static SHELL / SORTING... status screen
clears the counters
runs Shell sort
asserts ascending sortedness
displays the report if valid
```

The sort-status screen is deliberately static. It provides immediate visual feedback for long runs without adding progress counting, animation, key polling, or timing. Display work occurs before the counters are cleared and therefore remains outside the measured sorting experiment.

FK4 - VIEW

FK4 opens a deliberately small, read-only view of the current data array.

The screen displays up to four records, one per display row. Each line contains a 1-based, zero-padded decimal record number followed by as many leftmost key bytes as fit on the 32-column line. With the current display this is at most nine bytes:

```
0001: FF FF FF FF FF FF FF FF FF
```

For keys wider than the visible prefix, no horizontal scrolling is required.

Cursor Up and Cursor Down move the viewing window by one record. `EXE` returns to the ready state.

VIEW does not regenerate, edit, compare, or sort data. It displays whatever is currently present in the in-place data array:

- after startup or successful SETUP, the generated input;
- after Bubble or Shell completes, the resulting sorted array.

10. Post-sort assertion

After sorting, assert:

```
A[i-1] <= A[i]
```

for every adjacent pair from the beginning to the end of the array, using the same ascending binary ordering defined by this assignment.

Validation comparisons do not contribute to the sorting counters.

If the assertion fails, the run is invalid.

The normal report must not be displayed. The program shall report only the failed order assertion, for example:

```
ORDER=FAIL
```

A failed order assertion is considered a severe error requiring investigation.

11. Successful run report

After a valid sort, display a compact report containing:

```
algorithm
n
m
input pattern
KEY-CMP
INST-CMP
ORDER=PASS
```

The report may pause for `EXE` or an equivalent simple acknowledgement before returning to the ready state.

Returning from the report must leave the sorted data array intact so FK4 VIEW can inspect the result. The report itself remains about the experiment rather than about displaying individual records.

12. Explicit non-goals

The following are intentionally outside this assignment:

- random or pseudorandom input generation;
- general-purpose record browsing beyond the small read-only FK4 VIEW window;
- horizontal inspection of bytes beyond the visible leftmost record prefix;
- record editing;
- counting swaps;
- counting moves;
- timing measurements;
- stability guarantees;
- alternate Shell-sort gap sequences;
- general record payloads distinct from the key;
- larger experimental statistics beyond the two comparison counters.

These may become subjects of later CS617 chapters.

13. Acceptance criteria

Chapter 02 is successful when the PB-2000C program can:

1. accept valid `n`, `m`, and input-pattern parameters;

2. reject invalid configurations;
3. generate deterministic `ORDERED`, `REVERSE`, and `SAME` arrays;
4. sort those arrays in ascending binary order using optimized Bubble sort;
5. sort those arrays in ascending binary order using Shell sort;
6. perform both sorts in place;
7. report `KEY-CMP`, the algorithm-level key-comparison count;
8. report `INST-CMP`, the executed key-data comparison-instruction count;
9. exclude post-sort validation from the reported measurements;
10. regenerate the selected input before every sort;
11. assert sortedness after every run;
12. suppress the normal report and show `ORDER=FAIL` when the post-sort order assertion fails;
13. operate with an 8192-byte data-array capacity controlled by a single compile-time constant;
14. expose the current in-place data array through the read-only FK4 VIEW window without changing that data;
15. provide static algorithm-specific `SORTING...` feedback before the measured sort begins, without changing DATA or the comparison counters.

14. The boundary between WHAT and HOW

This assignment freezes the required behavior of the Chapter 02 sorting experiment.

The next phase is the **HOW**: mapping this contract onto the PB-2000C and HD61700, choosing memory placement, defining the comparator interface, constructing the setup form, and implementing and validating the two sorting algorithms.

Those implementation choices must preserve the behavior defined here.

A final thought

When you compare the numbers, resist the temptation to call the smaller one universally “faster.” These two counters measure specifically defined comparison work on controlled inputs. They do **not** measure elapsed time, total executed instructions, movement cost, cache effects, or all valid key distributions. A good report states exactly what was measured and what remains unknown.