Sorting algorithms: how are they affected by memory faults?

Alexandro Vladno
Centre for Informatics and Systems
University of Coimbra
Coimbra, Portugal
alexandro.vladno@gmail.com

Fabiano Papaiz
Centre for Informatics and Systems
University of Coimbra
Coimbra, Portugal
fabianopapaiz@gmail.com

Leo Moreira Silva
Centre of Informatics and Systems
University of Coimbra
Coimbra, Portugal
leo.moreira@me.com

Abstract

Although most people in the world use technology devices for many tasks, they don't know how the devices work and how they deal with faults. When those faults occur in memory, software behavior could be affected. Together with the software-specific algorithms are the sorting algorithms used to solve problems like ordering a list of products by their price. This work presents a discussion about how quicksort, mergesort, insertionsort and bubblesort algorithms are affected by memory faults.

Index Terms

sorting algorithms, memory faults.

I. Introduction

Technology is deeply introduced in people's quotidian supporting a massive number of tasks, for example: searching for a shared car, surfing on the web, sending a message to someone, automating the company's production or using the company's software. Nevertheless, most people don't know that devices are continually dealing with memory failures, faults and errors. These devices were made with large and inexpensive memories, which are also error-prone [1].

Software behavior may be affected by the problems mentioned before, especially those from memory. We have a memory fault when the correct value that should be stored in a memory location gets altered because of a soft failure. In particular, the content of a location can change unexpectedly, i.e., faults may happen at any time: real memory faults are indeed highly dynamic and unpredictable [2].

In the beginning steps of software development, the designer has a general idea of the structure and functions. For each one of these, some algorithms will be produced or used. In the following stages, the outcome software (and its algorithms) will be tested and, then, delivered to the user. Different kinds of algorithms could be written or used in the software, and one of these is the sorting algorithms.

A good algorithm is that which gives satisfactory results for every range of data set. Sorting is a fundamental concept and important for solving other problems like is prerequisite for Binary Search. Sorting is often used in a large variety of critical applications and is a fundamental task that is used by most computers [3].

In this paper, we present a discussion about how these sorting algorithms, particularly Quicksort, Mergesort, Insertion Sort and Bubblesort, are affected by memory faults.

II. Background

In this section we describe basic concepts about memory faults and sorting algorithms.

A. Memory Faults

Even the best digital system, with high-quality components and design techniques, may not be infallible to faults. Despite the title of this subsection, when the entire digital system (or software) is considered, there are three terms for computing fault and they have different meanings: failure, fault and error [4].

- *Error*: An error is a manifestation of a fault in a system, in which the logical state of an element differs from its intended value. An error occurs for a particular system state and input when an incorrect next state and/or output results.
- *Fault*: A fault is an anomalous physical condition. Causes include design errors, manufacturing problems, damage, fatigue, or other deterioration. Faults resulting from design errors and external factors are especially difficult to model and protect against because their occurrences and effects are hard to predict. A fault in a system does not necessarily result in an error;
- *Failure*: A failure denotes an element's inability to perform its functions because of error in the element itself or its environment, which in turn are caused by various faults;

B. Sorting Algorithms

Sorting algorithms are widely used in many aspects of data processing, information searches, business finance, computer encryption, etc. This work uses four sorting algorithms: quicksort, mergesort, insertionsort, and bubblesort. In the following subsections, we'll give an overview of them.

1) Quicksort: Quicksort algorithm, created by Hoare [5], is considered as one of the fastest and best sorting algorithms [6]. The algorithm is based on the paradigm of divide and conquer.

This algorithm has a execution time of $\theta(n^2)$ in the worst case over *n* numbers as input. Despite that execution time, quicksort is often the best option for sorting because of its remarkable average efficiency: $\theta(nlgn)$ [7].

The basic steps of this algorithm are [6]:

- Pick an element, which is called a pivot, from the list waiting to be sorted;
- Perform partition operation to realize that all elements in the list with values smaller than the pivot came before the pivot. Otherwise, all elements in the list with values bigger than the pivot come after it (elements which are equal to pivot can go either way). After this partition, the pivot is in the final position of the list;
- Recursively sort the sub-list of smaller elements and the sub-list of the bigger elements.

2) Mergesort: Mergesort was invented by John Von Newman and is one of the most elegant algorithms to appear in the sorting literature. It is the first sorting algorithm to have $\theta(nlgn)$ execution time bound. It is important to observe that this algorithm spends a lot of time on data transfer operations. In fact, standard Mergesort incurs about 2n data move operations [8].

Conceptually, Mergesort works as follows [8]:

- Divide the unsorted array into two sub arrays of about half the size;
- Sort each sub array recursively;
- Merge the two sub arrays back into one array.

3) Insertionsort: This algorithm sorts the array by shifting the elements one at time. It is efficient in sorting a small number of elements. The overall execution time of this algorithm is $\theta(n^2)$ [7]. The basic sorting steps are:

- If there are more than one element, pick the next element;
- Compare with all the elements in sorted sub-list;
- Shift all the elements in sorted sub-list that is greater than the value to be sorted;

- Insert the value;
- Repeat until list is sorted.

4) Bubblesort: The bubble sort is the oldest and simplest sorting method in use. It works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order) [9].

Table I below shows the time complexity comparison between the sorting algorithms presented. The *n* is the number of input elements.

TABLE I: Sorting algorithms complexity time comparison [10]

Algorithm	**Time Complexity**		
	Best Case	***Average Case***	***Worst Case***
Bubblesort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertionsort	$O(n)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(nlgn)$	$O(nlgn)$	$O(n^2)$
Mergesort	$O(nlgn)$	$O(nlgn)$	$O(nlgn)$

III. Materials and Methods

We first state our problem, then describe our data, showing all its characteristics. Then, we provide our hypothesis, define the setup, and perform the testing. Finally, we collect and perform data analysis.

A. Problem Statement

As introduced in the first section of this paper, sorting is a fundamental concept and essential for solving other problems. The content of memory location can change unexpectedly, i.e., faults may happen at any time. Considering this, the main objective of this work is to design experiments to answer the following question: *How are sorting algorithms affected by memory faults?*

B. Variables

For this experimental study, we assume that the independent and dependent variables are as shown in Table II and Table III below:

TABLE II: Independent variables.

Variable	**Description**
Probability of failure	Probability of a fault to occur
Array size	Size of the array of integers to be sorted
Sorting algorithm	Algorithm used to sort the array

TABLE III: Dependent variables.

Variable	Description
Largest subarray size	Size of the largest sorted subarray produced under the memory fault
Percentage of largest subarray size	Percentage of *largest subarray size* related to *array size* independent variable
Unordered elements quantity	Quantity of elements out of position after sorting algorithm execution. Adapted of the *k-unordered sequence* measure of disorder defined in [11]
Percentage of unordered elements quantity	Percentage of *unordered elements quantity* related to *array size* independent variable

C. Hypothesis

The set of hypothesis defined to test and draw some conclusions about this experiment are listed below. The confidence degree defined for hypothesis testing was 95% ($\alpha = 0.05$ and $\alpha - 1 = 0.95$).

- **Hypothesis 1:** For a given probability of failure and array size, tested algorithms will produce a different percentage of unordered elements quantity.
- **Hypothesis 2:** For a given probability of failure and array size, tested algorithms will produce a different percentage of the largest subarray size.
- **Hypothesis 3:** For each algorithm, the array size and probability of failure have a significative impact on the percentage of unordered elements quantity.
- **Hypothesis 4:** For each algorithm, the array size and probability of failure have a significative impact on the percentage of the largest subarray size.

D. Experimental Setup

To conduct the proposed study we get a set of files with the basic setup and instructions, that was composed by:

- A script that generates input files;
- Four files, one for each os those algorithms: quicksort, bubblesort, insertion sort, and mergesort, that are used to sorting the input data;

The input file looks like follow:

```
0.01 100 9 48 37 6 26 7 24 44 17 50 48 30 49 33 22 13 42 29 39 13 19 13 9 28
34 1 33 27 14 45 48 40 11 17 6 50 9 44 20 16 37 45 23 14 38 29 10 49 44 46 35
45 15 2 22 1 46 40 8 48 23 23 32 35 3 15 8 36 17 24 27 48 28 5 28 50 44 4 25
6 9 1 11 44 26 50 44 12 7 20 30 20 37 20 6 8 13 15 20 49
```

Fig. 1: Example of input file.

The input data shown in the Figure 1 is divided as follows:

- *Probability of Failure*: the first number of the sequence (`0.01`) is the probability of memory failure when sorting;
- *Sequence size*: the second number (`100`) means the size of the integers sequence used by sorting;
- *Sequence*: the rest of the numbers indicates the sequence itself.

We run 1000 times each algorithm with each file listed in Table IV below. We chose this number of executions to reduce possible noise in the generated data. Table IV shows 9 different input files with its sequence sizes and probabilities of failure. Plus, the number of executions and each sorting algorithm.

For example, Input A showed in that table with its characteristics was executed 1000 times for each algorithm - bubblesort, quicksort, mergesort, and insertion sort - totalizing 4000 executions.

TABLE IV: Generated input data.

Input	**Sequence Size**	**Probability of Failure**	**Executions**	**Algorithm**
Input A	100	1%	1000	Bubblesort Quicksort Mergesort Insertion Sort
Input B	100	2%	1000	Bubblesort Quicksort Mergesort Insertion Sort
Input C	100	5%	1000	Bubblesort Quicksort Mergesort Insertion Sort
Input D	1000	1%	1000	Bubblesort Quicksort Mergesort Insertion Sort
Input E	1000	2%	1000	Bubblesort Quicksort Mergesort Insertion Sort
Input F	1000	5%	1000	Bubblesort Quicksort Mergesort Insertion Sort
Input G	10000	1%	1000	Bubblesort Quicksort Mergesort Insertion Sort
Input H	10000	2%	1000	Bubblesort Quicksort Mergesort Insertion Sort
Input I	10000	5%	1000	Bubblesort Quicksort Mergesort Insertion Sort

Each execution generates an output file. Then, we generate 4000 outputs for each combination of sequence size, and the probability of failure, totalizing 36000 files. Figure 2 shows an output file.

```
[1]  9 48 37 6 26 7 24 44 17 50 48 30 49 33 22 13 42 29 39 13 19 13 9 28 34 1
33 27 14 45 48 40 11 17 6 50 9 44 20 16 37 45 23 14 38 29 10 49 44 46 35 45
15 2 22 1 46 40 8 48 23 23 32 35 3 15 8 36 17 24 27 48 28 5 28 50 44 4 25 6 9
1 11 44 26 50 44 12 7 20 30 20 37 20 6 8 13 15 20 49
[2]  1 1 1 2 3 4 5 6 6 6 6 7 7 8 8 8 9 9 9 9 10 11 11 12 13 13 13 13 14 14 15
15 15 16 17 17 17 19 20 20 20 20 22 22 23 23 23 24 24 25 26 26 27 27 28 28 28
29 29 30 30 32 33 33 34 35 35 36 37 37 37 38 39 40 40 42 44 44 44 44 44 44 20
45 45 45 46 46 48 48 48 48 48 49 49 50 49 50 50 50
[3]  1 1 1 2 3 4 5 6 6 6 6 7 7 8 8 8 9 9 9 9 10 11 11 12 13 13 13 13 14 14 15
15 15 16 17 17 17 19 20 20 20 20 20 22 22 23 23 23 24 24 25 26 26 27 27 28 28
28 29 29 30 30 32 33 33 34 35 35 36 37 37 37 38 39 40 40 42 44 44 44 44 44 44
45 45 45 46 46 48 48 48 48 48 49 49 49 50 50 50 50
[4]  82
```

Fig. 2: Example of output file.

The output file gives four essential data, as enumerated below:

- *[1]*: the original sequence of integers contained in the input file;
- *[2]*: the sequence processed by the sorting algorithm under the memory fault model;
- *[3]*: the sequence sorted correctlty;
- *[4]*: the size of the largest sorted subsequence in [2]. This number can be interpreted as the quality of sorting. As higher, most successful was the sorting operation.

E. Development

We develop Python scripts to generate our dataset. First, we generate all input files through a given script called *gen.py*. This script considers sequence size and the probability of failure to create a sequence of integers. Were produced nine input files like Figure 1, combining the sequence size and probability of failure listed in Table IV.

After that, we developed a script called *main_execution.py* that is responsible for creating output files based on the input files exposed before. At this time, for every input showed in Table IV we ran 1000 times to minimize any deviation on the quality of sorting indicator (line *[4]* on Figure 2).

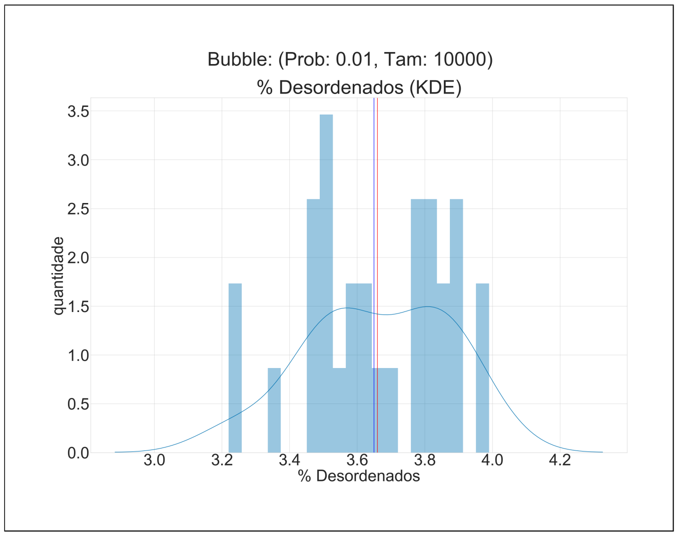
As said before, after the execution of this algorithm, 4000 outputs for each combination of sequence size, and the probability of failure, totalizing 36000 files.

To be continued...

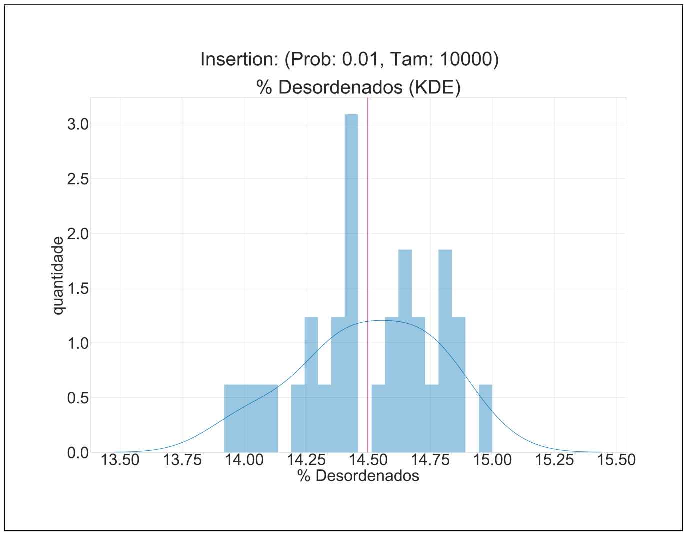
F. Data Analysis

In this section, we describe our data and the statistic tests used to test the hypothesis showed before.

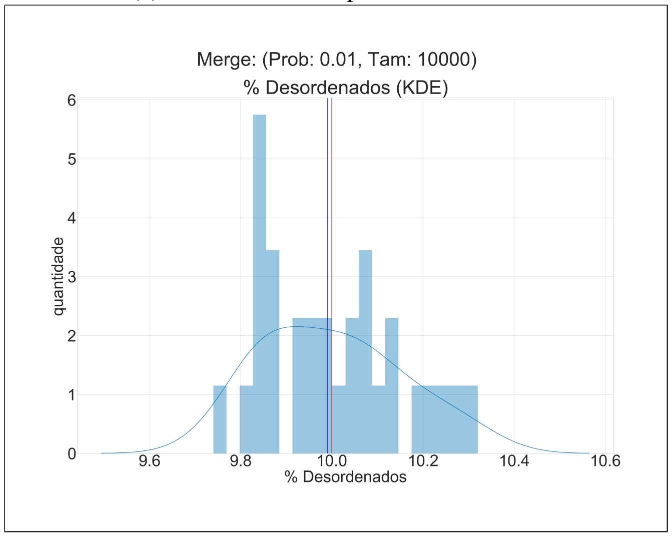
1) Exploratory Data Analysis (EDA): Based on the output files generated before, our starting point was to test if the dataset had a normal distribution. To achieve this, we use Shapiro-Wilk normality test. After we run these tests, only the distribution related to the dependent variable *unordered elements size* was considered normal. Figure 3 shows an example of this distribution. In these graphs, the red vertical line means the mean, and the blue vertical line means the median. The distributions were normal in all combinations of independent variables for dependent variable *unordered elements size*.



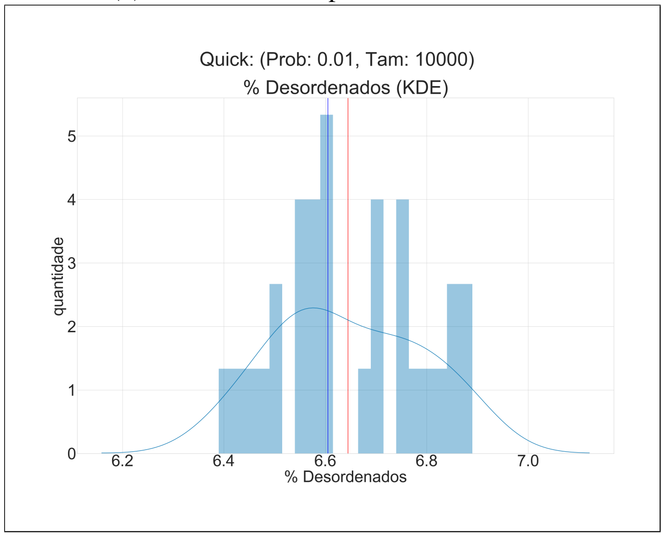
(a) Distribution Graph for bubblesort.



(b) Distribution Graph for insertion sort.



(c) Distribution Graph for mergesort.



(d) Distribution Graph for quicksort.

Fig. 3: Distribution graph for a probability failure of 1% and a sequence size of 10000.

Q-Q plot shows that how much more blue points close to the red line, most normal is the distribution. Figure 4 below presents this graph for a probability failure of 1% and a sequence size of 10000 for considered sorting algorithms.

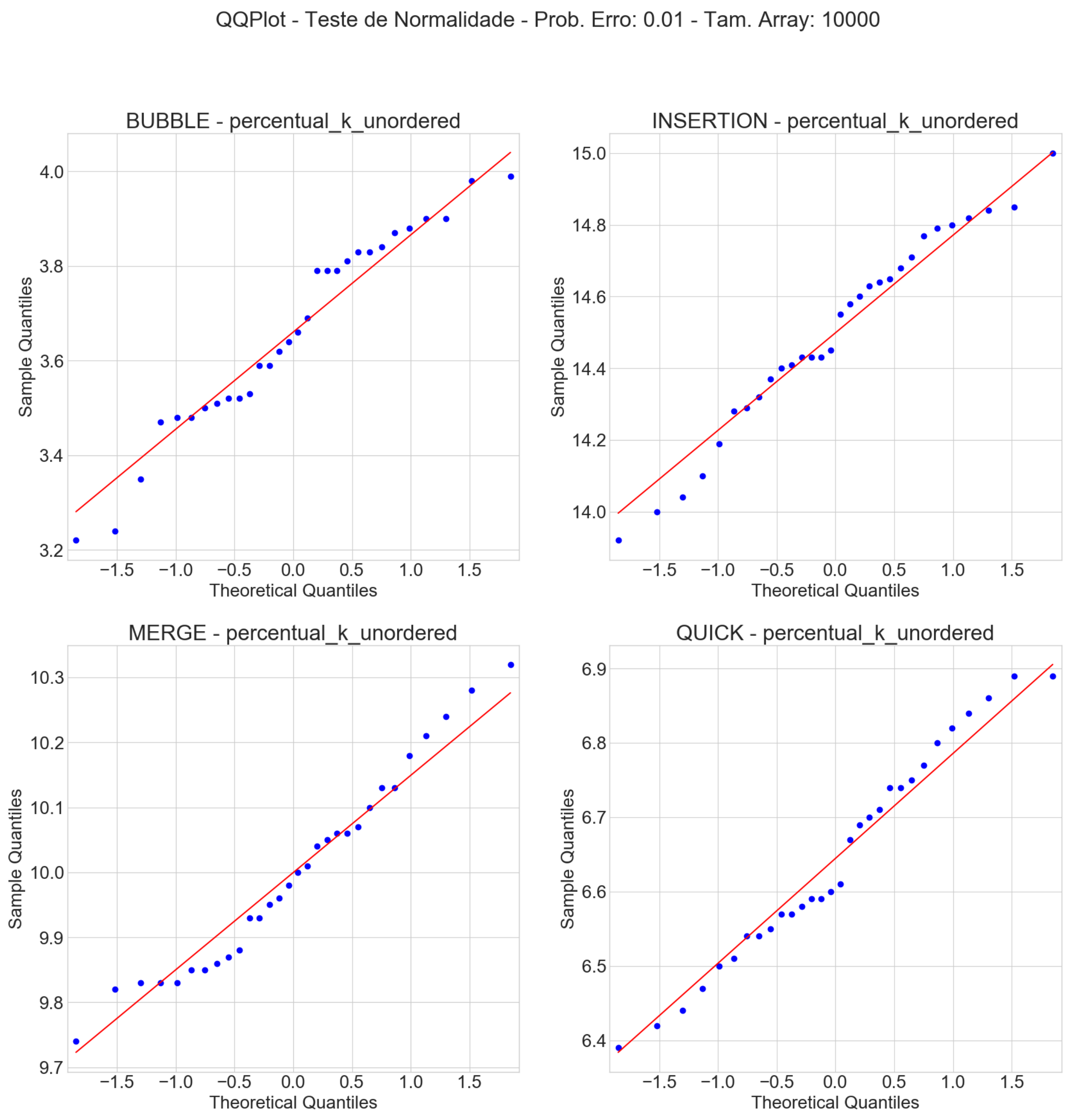


Fig. 4: Q-Q plot showing normal distribution for a probability failure of 1% and a sequence size of 10000.

G. *Conclusions*

IV. Results

V. Discussion

VI. Conclusion

VII. Prepare Your Paper Before Styling

Before you begin to format your paper, first write and save the content as a separate text file. Complete all content and organizational editing before formatting. Please note sections VII-A–VII-E below for more information on proofreading, spelling and grammar.

Keep your text and graphic files separate until after the text has been formatted and styled. Do not number text heads—LaTeX will do that for you.

A. *Abbreviations and Acronyms*

Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, ac, dc, and rms do not have to be defined. Do not use abbreviations in the title or heads unless they are unavoidable.

B. *Units*

- Use either SI (MKS) or CGS as primary units. (SI units are encouraged.) English units may be used as secondary units (in parentheses). An exception would be the use of English units as identifiers in trade, such as "3.5-inch disk drive".
- Avoid combining SI and CGS units, such as current in amperes and magnetic field in oersteds. This often leads to confusion because equations do not balance dimensionally. If you must use mixed units, clearly state the units for each quantity that you use in an equation.
- Do not mix complete spellings and abbreviations of units: "Wb/m^2" or "webers per square meter", not "webers/m^2". Spell out units when they appear in text: ". . . a few henries", not ". . . a few H".
- Use a zero before decimal points: "0.25", not ".25". Use "cm^3", not "cc".)

C. *Equations*

Number equations consecutively. To make your equations more compact, you may use the solidus (/), the exp function, or appropriate exponents. Italicize Roman symbols for quantities and variables, but not Greek symbols. Use a long dash rather than a hyphen for a minus sign. Punctuate equations with commas or periods when they are part of a sentence, as in:

$$a + b = \gamma \tag{1}$$

Be sure that the symbols in your equation have been defined before or immediately following the equation. Use "(1)", not "Eq. (1)" or "equation (1)", except at the beginning of a sentence: "Equation (1) is . . ."

D. *LaTeX-Specific Advice*

Please use "soft" (e.g., `\eqref{Eq}`) cross references instead of "hard" references (e.g., `(1)`). That will make it possible to combine sections, add equations, or change the order of figures or citations without having to go through the file line by line.

Please don't use the `{eqnarray}` equation environment. Use `{align}` or `{IEEEeqnarray}` instead. The `{eqnarray}` environment leaves unsightly spaces around relation symbols.

Please note that the `{subequations}` environment in LaTeX will increment the main equation counter even when there are no equation numbers displayed. If you forget that, you might write an article in which the equation numbers skip from (17) to (20), causing the copy editors to wonder if you've discovered a new method of counting.

BIBTeX does not work by magic. It doesn't get the bibliographic data from thin air but from .bib files. If you use BIBTeX to produce a bibliography you must send the .bib files.

LaTeX can't read your mind. If you assign the same label to a subsubsection and a table, you might find that Table I has been cross referenced as Table IV-B3.

LaTeX does not have precognitive abilities. If you put a `\label` command before the command that updates the counter it's supposed to be using, the label will pick up the last counter to be cross referenced instead. In particular, a `\label` command should not go before the caption of a figure or a table.

Do not use `\nonumber` inside the `{array}` environment. It will not stop equation numbers inside `{array}` (there won't be any anyway) and it might stop a wanted equation number in the surrounding equation.

E. Some Common Mistakes

- The word "data" is plural, not singular.
- The subscript for the permeability of vacuum μ_0, and other common scientific constants, is zero with subscript formatting, not a lowercase letter "o".
- In American English, commas, semicolons, periods, question and exclamation marks are located within quotation marks only when a complete thought or name is cited, such as a title or full quotation. When quotation marks are used, instead of a bold or italic typeface, to highlight a word or phrase, punctuation should appear outside of the quotation marks. A parenthetical phrase or statement at the end of a sentence is punctuated outside of the closing parenthesis (like this). (A parenthetical sentence is punctuated within the parentheses.)
- A graph within a graph is an "inset", not an "insert". The word alternatively is preferred to the word "alternately" (unless you really mean something that alternates).
- Do not use the word "essentially" to mean "approximately" or "effectively".
- In your paper title, if the words "that uses" can accurately replace the word "using", capitalize the "u"; if not, keep using lower-cased.
- Be aware of the different meanings of the homophones "affect" and "effect", "complement" and "compliment", "discreet" and "discrete", "principal" and "principle".
- Do not confuse "imply" and "infer".
- The prefix "non" is not a word; it should be joined to the word it modifies, usually without a hyphen.
- There is no period after the "et" in the Latin abbreviation "et al.".
- The abbreviation "i.e." means "that is", and the abbreviation "e.g." means "for example".

An excellent style manual for science writers is [**?**].

F. Authors and Affiliations

The class file is designed for, but not limited to, six authors. A minimum of one author is required for all conference articles. Author names should be listed starting from left to right and then moving down to the next line. This is the author sequence that will be used in future citations and by indexing services. Names should not be listed in columns nor group by affiliation. Please keep your affiliations as succinct as possible (for example, do not differentiate among departments of the same organization).

G. Identify the Headings

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

Component heads identify the different components of your paper and are not topically subordinate to each other. Examples include Acknowledgments and References and, for these, the correct style to use is "Heading 5". Use "figure caption" for your Figure captions, and "table head" for your table title. Run-in heads, such as "Abstract", will require you to apply a style (in this case, italic) in addition to the style provided by the drop down menu to differentiate the head from the text.

Text heads organize the topics on a relational, hierarchical basis. For example, the paper title is the primary text head because all subsequent material relates and elaborates on this one topic. If there are two or more sub-topics, the next level head (uppercase Roman numerals) should be used and, conversely, if there are not at least two sub-topics, then no subheads should be introduced.

H. Figures and Tables

a) Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation "Fig. **??**", even at the beginning of a sentence.

TABLE V: Table Type Styles

Table Head	**Table Column Head**		
	Table column subhead	***Subhead***	***Subhead***
copy	More table copy[a]		

[a]Sample of a Table footnote.

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity "Magnetization", or "Magnetization, M", not just "M". If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write "Magnetization (A/m)" or "Magnetization {A[m(1)]}", not just "A/m". Do not label axes with a ratio of quantities and units. For example, write "Temperature (K)", not "Temperature/K".

ACKNOWLEDGMENT

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks …". Instead, try "R. B. G. thanks…". Put sponsor acknowledgments in the unnumbered footnote on the first page.

REFERENCES

[1] I. Finocchi and G. F. Italiano, "Sorting and searching in the presence of memory faults (without redundancy)," in *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, no. January, pp. 101–110, 2004.

[2] S. Hamdioui, Z. Al-Ars, A. J. Van De Goor, and M. Rodgers, “Dynamic faults in random-access-memories: Concept, fault models and tests,” in *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 19, pp. 195–205, 2003.

[3] NitinArora and S. Kumar, “A Novel Sorting Algorithm and Comparison with Bubble sort and Insertion sort,” *International Journal of Computer Applications*, vol. 45, no. 1, p. 2.

[4] V. P. Nelson, “Fault-Tolerant Computing: Fundamental Concepts,” *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[5] C. A. R. Hoare, “Quicksort,” *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[6] X. Wang, “Analysis of the time complexity of quick sort algorithm,” *Proceedings - 2011 4th International Conference on Information Management, Innovation Management and Industrial Engineering, ICIII 2011*, vol. 1, pp. 408–410, 2011.

[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms.* MIT Press, 2009.

[8] D. Abhyankar and M. Ingle, “A Novel Mergesort,” vol. 1, no. 3, pp. 17–22, 2011.

[9] K. Mansotra, V; Sourabh, “Implementing Bubble Sort Using a New Approach,” pp. 1–6, 2011.

[10] P. Prajapati, N. Bhatt, and N. Bhatt, “Performance Comparison of Different Sorting Algorithms,” vol. VI, no. Vi, pp. 39–41, 2017.

[11] U. Ferraro-Petrillo, I. Finocchi, and G. F. Italiano, “The price of resiliency: A case study on sorting with memory faults,” *Algorithmica (New York)*, vol. 53, no. 4, pp. 597–620, 2009.