Writing a report on experiments with algorithms

Herman Haverkort
Eindhoven University of Technology[*]

DBL algorithms (2IO90), spring 2013

About this document

This document is a part of a document written by Herman Haverkort for the instance of *DBL Algorithms* that ran in Spring 2013. In that instance the task was to design algorithms for clustering a set of points in the plane. However, the writing tips in this document are also useful for the current instance of *DBL Algorithms.* **Note: in case this documents contains information that contradicts the information in other documentation for the current instance, the latter information has precedence.**

Contents

1 **Contents of the report** *4*
1.1 *Abstract* *4*
1.2 *Introduction: describing the problem* *4*
1.3 *Describing algorithms* *5*
1.3.1 Describing input and output *5*
1.3.2 Describing the algorithm *5*
1.3.3 Making things precise *6*
1.3.4 Analyzing the output quality of your algorithm *7*
1.3.5 Analyzing the efficiency (running time and memory usage) *7*
1.3.6 Formulating theorems *7*
1.4 *Notation* *8*
1.5 *Writing proofs* *9*
1.5.1 Equivalence proofs *9*
1.5.2 Proof by contradiction *9*
1.6 *Describing and discussing experiments* *9*
1.6.1 Describing the test environment *9*
1.6.2 Describing the test data *10*
1.6.3 Describing the algorithms tested *10*
1.6.4 Describing the experiments and the results *10*
1.6.5 Discussing the results *10*
1.7 *Conclusions/discussion/recommendations* *11*

[*] Some material in this document may have been taken from several texts for classroom use by several current and former instructors of the Eindhoven University of Technology, including Mark de Berg, Bettina Speckmann, and Alexander Wolff.

1.8 *Acknowledgements* *11*
1.9 *Bibliography* *12*
1.10 *Structure* *13*
1.10.1 Sections and subsections *13*
1.10.2 Paragraphs *13*
2 **Lay-out and typography** ***14***
2.1 *(No) Cover page* *14*
2.2 *Titles of the document, sections, subsections and paragraphs* *14*
2.3 *Text* *14*
2.4 *Bibliography* *14*
3 **Language** ***15***
3.1 *Style: personal versus impersonal* *15*
3.1.1 You becomes we *15*
3.1.2 I becomes we *15*
3.1.3 It becomes we *15*
3.1.4 Her, him, and it *15*
3.2 *Style: variation versus monotonicity* *16*
3.3 *Grammar and structure of sentences* *16*
3.3.1 Length of sentences *16*
3.3.2 Much, many, amount, and number *16*
3.3.3 Then and than *16*
3.3.4 To be done and to have been done *17*
3.3.5 References *17*
3.3.6 Type consistency *17*
3.4 *Words and concepts to be careful with* *17*
3.4.1 Miljard, billion etc. *17*
3.4.2 Maximum and maximal *18*
3.4.3 Worst-case complexity *18*
3.4.4 Average-case complexity *18*
3.4.5 Best-case complexity *18*
3.4.6 Expected running time *18*
3.4.7 It is easy to see *19*
3.4.8 Respectively *19*
3.5 *Spelling* *19*
3.6 *Numbers* *19*
3.7 *Punctuation* *20*
3.7.1 Commas (,) *20*
3.7.2 Colons (:) versus commas *20*
3.7.3 Semicolons (;) versus commas *20*
3.7.4 Apostrophes (') *21*
3.7.5 Hyphens *21*
3.8 *Typesetting formulas* *21*
3.9 *Mixing mathematical symbols and text* *22*
4 **Pseudocode** ***23***
4.1 *Loops* *23*
4.2 *Branching* *24*
4.3 *Operators* *24*
4.4 *Variables and arrays* *24*
4.5 *Function calls* *25*

4.6 *Comments* *25*
4.7 *Length and integration in the text* *25*
5 **Charts** ***26***
6 **Tables** ***28***

1 Contents of the report

1.1 Abstract

In the abstract you give an overview, typically one short paragraph, of the contents of your report: you describe the problem you have studied, and what the main results are. Make it as concise as possible. To motivate your studying the problem it suffices to mention an application area (for example, "data mining") with one word.

1.2 Introduction: describing the problem

The introduction should be written such that it can be understood without reading another document first. In particular, you cannot assume the reader to be familiar with the problem description that was provided to you at the beginning of this DBL course. Take the introduction very seriously: this should be the best-written part of the whole report. For your report to be convincing, the reader has to be willing to believe that you implemented your algorithms and experiments correctly, so you really have to convince the reader that you know what you are doing.

The introduction usually starts with a description of and a motivation for the general problem area. In this project, that could be data mining. You may refer to previous or related work here. This way you can explain why the problem is interesting.

After introducing the general problem area, you zoom in to the specific problem studied in the report. Then you change to a more abstract point of view, describing the problem in an abstract way that omits all details and context that are irrelevant for the study of its algorithmic solution. For example, whereas before you might have been writing about records in a database, you may now limit the discussion to a problem on points in the plane. Describe what is the input to your computational problem, and what properties the output of your computation should have.

You may refer to more previous work on the abstract problem here. You may also mention any previous work on very similar problems here, and mention whether or not the algorithms from that previous work are applicable to your problem.

Ideally, the discussion of the previous work culminates in a clear statement about what is still missing in the current state-of-the-art (namely an answer to the specific problem you study), and why (what are the principal problems to overcome).

Then you give an overview of your results. Try to include a high-level, but comprehensible, description of the approach(es) you have used to obtain those results, and relate them to approaches found in the literature. State any theoretical guarantees (on running time, for instance, or on other aspects) that you may have proved for your algorithms and mention the main conclusions from the experiments. Include references to sections of the rest of your report, where the results are explained in detail.

1.3 Describing algorithms

1.3.1 Describing input and output

Before describing an algorithm, specify clearly:

- what is the input to your algorithm; what properties does it have?
- what is the output of your algorithm; what properties should it certainly have (strict requirements)?
- if multiple valid outputs are possible, what properties of the output should your algorithm try to realize (soft requirements) or optimize (optimization criteria)?

Most algorithms should not ignore the input: the output should somehow depend on the input. Therefore, check that the requirements on the output are formulated such that they depend on the input.

1.3.2 Describing the algorithm

Your main challenge is now to describe the algorithm such that its description is not only complete, accurate, and implementable, but also understandable to human beings. Usually it is best to first describe the basic idea of the algorithm on a conceptual level, without implementation details. Make sure that this description can be understood without reading the details first: you should not refer to concepts that will only be explained later.

The high-level conceptual description is typically followed by a number of paragraphs that describe each step of the algorithm and each of the necessary data structures in detail. Make sure you give plenty of "intuition": describe what is the meaning and purpose of all steps in the algorithm and describe what operations should be performed efficiently by supporting data structures. Alternatively, you can develop the algorithm in an incremental way: start from an algorithm that is simple to understand but not good enough (for example, too slow, or not entirely correct); then describe how to improve the algorithm step by step to obtain an algorithm that is efficient and fully correct. In any case, you do not need to describe the details of standard data structures from the literature: it is sufficient to explain how exactly you employ the data structures in your algorithm, and include a reference to the literature where more details about the data structure can be found.

After describing the details of a complicated algorithm, the reader may be overwhelmed and unsure about how everything fits together. This is especially true if you described the algorithm in an incremental way. You can make sure the reader is still with you by finishing with a concise description of the complete algorithm from beginning to end, this time with most details in, but without explaining why things as they are and without arguing why the algorithm is correct. This concise description could also be given in pseudocode (see Section 4).

At any stage in the description (initial conceptual description, details, final concrete description) you may use pseudocode to clarify your explanation, and to show how to deal with some special cases that would not be interesting enough to describe in the main text. However, do not use pseudocode as a replacement for normal text: your explanation should still make sense if the reader decides to skip all pseudocode.

1.3.3 Making things precise

Throughout the description your algorithm, its input, its output, and its analysis, you should define all concepts you use clearly and precisely.

Use variables to give names to mathematical objects that are referred to later, so that you can refer to them clearly and unambiguously:

> ~~The algorithm iterates over all vertices. For each vertex, it checks if it is has a neighbour such that there is another vertex that is adjacent to this vertex and there is a path from the original vertex to this other vertex that does not contain the second vertex. If so, we put the original vertex in the queue.~~
> The algorithm iterates over all vertices. For each vertex u, it checks if there are vertices $v, w \neq u$ such that v is a neighbour of both u and w, and there is a path from u to w that does not contain v. If such vertices v, w exist, then we put u in the queue.

Being precise does not necessarily mean that you have to use mathematical formulas. Often a definition in normal English language, without formulas, is much clearer and equally precise. Consider as an example the following problem:

> Let R be a set of axis-parallel rectangles in the plane. For a rectangle $r_i \in R$, let x_i be the x-coordinate of its left edge, and let x'_i be the x-coordinate of its right edge. Similarly, let y_i and y'_i be the y-coordinates of the bottom and top edge of r_i, respectively. We want to find a subset $R' \subset R$ such that

$$\begin{gathered}\forall(r_i, r_j) \in R' \times R' : (x'_i < x_j) \vee (x_i > x'_j) \vee (y'_i < y_j) \vee (y_i > y'_j) \\ \wedge \\ \forall R'' \subset R : \left(\left(\forall(r_i, r_j) \in R'' \times R'' : (x'_i < x_j) \vee (x_i > x'_j) \vee (y'_i < y_j) \vee (y_i > y'_j) \right) \Rightarrow |R''| \leq |R'| \right) .\end{gathered}$$

Compare this to the following definition of the same problem:

> Let R be a set of axis-parallel rectangles in the plane. We want to find a subset $R' \subset R$ with a maximum number of rectangles under the condition that no two rectangles in R' intersect each other. The rectangles in R are defined to be closed, so two rectangles are considered to intersect each other even if only their boundaries intersect[1].

Of course, if you want to implement your algorithm, you may need to work out formulas that specify when two rectangles r_i and r_j are disjoint: $(x'_i < x_j) \vee (x_i > x'_j) \vee (y'_i < y_j) \vee (y_i > y'_j)$. But this is something readers should be able to figure out for themselves. The space in your report is better used to explain things that are more complicated or more interesting.

Use figures to illustrate the input and output of your algorithm, and to illustrate the definitions of concepts (variables) you use to describe the algorithm and to prove its correctness. However, figures cannot *replace* definitions in words. A printed figure can only show one concrete instance of all the possible configurations of (geometric) objects covered by your definition. From a figure, the reader can never tell for sure which other configurations are covered by your definition, so a definition in words is still needed. Furthermore, make sure your figure does not show a special case from which the reader would be tempted to infer properties that are not true in general. For example, if you want to show a general set of points, do not put all of them on one line.

1. This example is due to Mark de Berg

1.3.4 Analyzing the output quality of your algorithm

The description of the algorithm should already explain what is the purpose and meaning of each step in the algorithm. In some cases this already makes it clear that the algorithm is correct. In other cases, a proof of correctness is needed. See Section 1.5 for some basic advice on how to write proofs.

If your algorithm is an optimization algorithm, you should try to analyze the quality of the output. Can the quality of the output be measured by a well-defined quality measure? Does your algorithm return optimal output according to this measure, or close-to-optimal output? Carefully distinguish between what you can prove and what you hope to be true. "The algorithm returns an optimal clustering" is not the same as "We did our best to make the algorithm do things that result in good clusterings and it seems to work". To an algorithms researcher, "The algorithm returns an optimal clustering", means: "It can be *proven* that the algorithm *always* returns an optimal clustering", and if you make such a claim, you should prove it.

If you cannot prove that the algorithm returns an optimal solution, then try to give some examples of inputs on which the algorithm should typically be expected to perform well, and of inputs on which the algorithm may perform badly.

1.3.5 Analyzing the efficiency (running time and memory usage)

Typically, the description of the algorithm does not mention running times of individual steps, because this would be too distracting: when reading the description of the algorithm, the reader needs to concentrate on understanding what the algorithm does and why it works. Therefore it is usually best to defer the analysis of the running time and memory usage of an algorithm completely until after the final description of the complete algorithm. Make sure that all variables you use in the analysis are well-defined: do not forget to define n.

1.3.6 Formulating theorems

You may summarize the theoretical properties of your algorithm in a *theorem*. This is especially appropriate if these properties belong to the main results of your work. A typical theorem about an algorithm has the following form:

Theorem 1 *There is an algorithm that, given an (input) of n (things), computes an (output) that satisfies (some conditions depending on the input) in $O($some function of $n)$ time and $O($some function of $n)$ memory.*

Indicate clearly what part of your text constitutes the proof of your theorem. Typically this is done in one of the following ways:

- (only if the proof is short) state the theorem, start the proof with "*Proof:*" and conclude the proof with "$\square$";
- state the theorem, write "We will now prove this theorem", give the proof, and conclude with "This concludes the proof of Theorem (*number*)";
- give the proof, write "This leads to the following theorem:", and state the theorem.

1.4 Notation

To be able to describe your algorithms and/or state theorems accurately and concisely, you may need to introduce some notation. Choose notation that is easy to read and easy to remember. Here are some guidelines for how to accomplish this.

Use similar notation for similar concepts, and follow conventions as much as possible. What works best depends on the needs of your text, but here are some ideas:

- lowercase greek letters for real numbers and angles; in particular, ϵ and δ are often used for *small* real numbers;
- lowercase latin letters for natural numbers and basic objects such as points (p, q), vertices (u, v, w), lines (ℓ); in particular, i and j are usually indices of elements in a vector (array) or a matrix; input size is usually denoted by n; in the case of graphs, n is usually the number of vertices and m is usually the number of edges;
- capital letters for sets;
- calligraphic letters for sets of sets and for structures such as graphs $(\mathcal{G})$ and trees $(\mathcal{T})$.

Do not make notation more complicated than necessary. For example, you may define $\delta_{\mathcal{G}}(u, v)$ to be the distance from vertex u to vertex v in the weighted graph $\mathcal{G}$. But if, for any pair of vertices u and v, it is always clear which graph is meant, then the notation $\delta(u, v)$ will do just fine.

You have some freedom to choose between parentheses, subscripts and superscripts to identify properties of objects. Compare:

- Let $p_1, ..., p_n$ be the vertices of a polygon P. We denote the x-coordinate of p_i by x_{p_i}.
- Let $p_1, ..., p_n$ be the vertices of a polygon P. We denote the x-coordinate of p_i by $x(p_i)$.
- Let $p_1, ..., p_n$ be the vertices of a polygon P. We denote the x-coordinate of p_i by x_i.

Avoid the first solution: the second subscript is so small that it is barely readable. The second solution is acceptable, but, when used in a complex formula that contains many parentheses already, this solution may burden the reader with even more nested parentheses to match. The third solution may therefore be preferable, but it may not always be possible (for example, what if you are discussing two polygons P and Q with vertices $p_1, ..., p_m$ and $q_1, ..., q_n$?)

Use short names only for concepts that are only defined and referred to within one subsection or proof. Use longer, more descriptive names for concepts that are used throughout the document. *Names of variables and functions can be longer than one letter.* The notation "$\mathit{weight}(v)$" is clearer than "$w(v)$". Compare also:

> ~~Given a set of points P and a line ℓ, let P_1, P_2, and P_3 be the sets of points to the left of ℓ, on ℓ, and to the right of ℓ, respectively.~~
>
> Given a set of points P and a line ℓ, let $P_{<\ell}$, P_ℓ, and $P_{>\ell}$ be the sets of points to the left of ℓ, on ℓ, and to the right of ℓ, respectively.

Think about where you want to introduce your notation. In principle any notation should be introduced at the highest level in the document that includes all places where you need to use the notation: if you need the notation in only one paragraph, define it in that paragraph; if you need the notation in two paragraphs of the same (sub)section, define it in that (sub)section; if you need the notation in multiple sections of the document, define it in the introduction (or in a separate section after the introduction). No matter where you define your notation, the reader is likely to forget it if it has not been used for more than a page, or if the reader has been doing something else and resumes reading your report in the middle. It can be very helpful to the reader if you occasionally repeat a definition just before it is used, by writing things like: "Recall that $\delta(u, v)$ is the distance from u to v in $\mathcal{G}$."

1.5 Writing proofs

You may need to prove properties of your problem or your algorithms in your report. Below are a few tips for some types of proofs you may need to write.

1.5.1 Equivalence proofs

To prove that a statement A is true if and only if a statement B is true, you always need to prove two things. Prove one of the following combinations:

- (i) if A is true, then B is true; (ii) if B is true, then A is true
- (i) if A is true, then B is true; (ii) if A is not true, then B is not true
- (i) if B is not true, then A is not true; (ii) if B is true, then A is true
- (i) if B is not true, then A is not true; (ii) if A is not true, then B is not true

Make it clear that you will prove both parts, and make clear which part of your proof proves what:

> **Theorem 2** *A triangle T is equilateral if and only if all angles of T are $\pi/3$.*
>
> *Proof:* We will have to prove that (i) if T is equilateral, all angles of T are $\pi/3$, and (ii) if all angles of T are $\pi/3$, then T is equilateral.
> To prove (i), ... This concludes the proof of part (i).
> To prove (ii), ... This concludes the proof of part (ii). □

1.5.2 Proof by contradiction

The typical form of a proof by contradiction is: "*For the sake of contradiction, assume* (assumption A, which is the opposite of want we want to prove) (reasoning leading to conclusion B)*. However, by our assumption that we have* (assumption A)*, we have* (reasoning leading to the conclusion that B is false)*. This contradicts* (conclusion B)*; hence,* (assumption A) *is false and* (what we want to prove) *is true.*

1.6 Describing and discussing experiments

A description of experiments and results should include the following:

- a description of the test environment
- a description of the test data
- a description of the algorithms tested
- a presentation of the experiments and results
- a discussion of the results

1.6.1 Describing the test environment

Describe the test environment to the extent that is relevant for your experiments. If running time matters, mention what machine was used (processors, amount of memory), and what programming language was used.

1.6.2 Describing the test data

Describe how the test data was obtained:

- Specify what files were downloaded from what sources: give enough details so that others could repeat your experiments with the same results.
- Specify how random data was generated. You should always give enough detail (include all parameter settings) so that a skilled programmer could implement an algorithm that generates random data with the same properties as your random data.

Give data sets clear names that could be used as labels in charts and tables. Include figures to give the reader an impression what each data set looks like. With big data sets it is often impossible to print a figure of the complete data set: in that case you can show a small part of it, or a random sample of it (indicate clearly what you do).

1.6.3 Describing the algorithms tested

You probably described your algorithms earlier in the report already. Sometimes these descriptions still left things open (implementation choices, parameter settings) that were irrelevant to the theoretical description of your algorithm, but could nevertheless influence the results of your experiments. Describe all such details: give enough detail so that a skilled programmer could implement your algorithms, repeat your experiments, and obtain the same results.

Give algorithms clear names that could be used as labels in charts and tables.

1.6.4 Describing the experiments and the results

Present the results of your experiments in charts and/or tables. Make sure that for each experiment it is clear what test data was used, what exact algorithm was tested, and how your algorithm was evaluated. If the algorithm is non-deterministic (making random choices), specify how many times you ran each algorithm on each data set, and how you aggregated the results (for example: did you take the average, the best result, or the worst result).

1.6.5 Discussing the results

Describe the general trends that can be observed in your charts and tables, as well as outliers (results that do not correspond to the general trends, or results that are somehow suspicious). Check and double-check that your observations are clearly visible in the data you present in charts and tables. If you cannot find a reasonable way to draw the charts such that the trends are clearly visible and can be verified by the reader, then you should ask yourself if the supposed trend is not merely a product of wishful thinking.

Discuss how your experimental results relate to the theoretical analysis and expectations you had of your algorithm: do the results confirm your expectations, or do they contradict your expectations?

1.7 Conclusions/discussion/recommendations

In the last section of your report, you relate your results to the problem statement in the introduction. This means you have to do more than simply copying your results from the abstract, the introduction, and the rest of your report. The last section of your report should clarify *to what extent* you solved the problem presented in the introduction. Ask yourself the following questions:

- In what aspects and for what types of input is your algorithm very good?
- In what aspects and for what types of input is your algorithm not good?
- Do you have an idea what may be the cause of bad performance of your algorithm on certain types of input?
- If yes, how could you (or somebody else, in future research) test if this idea is correct?
- What would be the main problem to overcome if you want to get a better algorithm?
- Could your algorithm also be used to solve other (possibly similar) problems that are beyond the scope of your original problem statement?

Any claims you make should be supported by the material presented in the report before. Keep a clear distinction between claims for which you have a theoretical proof, claims for which you have evidence, and claims which may be true by lack of evidence to the contrary. "Our algorithm always produces the correct result" should mean that for this claim, you have a theoretical proof that is valid for all possible inputs. If you do not have theoretical proof, you may be able to write: "*In our experiments*, our algorithm always produced the correct result".

In your evaluation, you may also consider aspects of the algorithm that may not be covered by the problem statement, but in hindsight, should have been. For example, if your algorithm solves the problem as presented in the introduction perfectly, but in hindsight in turns out that the output is useless for practical applications, then there was something wrong with the problem statement. In that case, you could discuss how the problem statement might be improved.

Note that lazy readers only read the introduction and the conclusions of a report. This is where you can show that you really have a good understanding of the topic.

1.8 Acknowledgements

Many reports have a brief section called "Acknowledgements". Here you acknowledge any type of help you got from other people, for example, useful comments, software tools or test data that are not publicly available, help with technical problems etc.

1.9 Bibliography

At the end of the report, there is a bibliography or reference list. Here you list all written sources that are referred to in the main text and give more details on the problem area, previous work, data structures used etc. The main "rules" for the bibliography are:

- only list sources that are *explicitly referred to* in the main text (this rule is strict);
- *verify* that each of these sources gives the information for which it is referred to;
- give the *most reliable* sources;
- try to use sources that are *easily accessible*;
- give enough *publication details* so that it is absolutely unambiguous which source (and which version of it) you refer to, and so that the reader can retrieve it.

Verifying the source means that each item in the bibliography must have been seen by at least one author of the report with her own eyes. The source must be written in a language that is understood by the author who verified the source.

Sources can be, from generally reliable to generally unreliable:

- well-known text books and publications in well-known international academic journals;
- PhD theses;
- long papers (more than six pages) in peer-reviewed international conferences;
- technical reports;
- any other printed publications;
- websites.

If you used knowledge from lesser sources, always try to replace them by more reliable sources. Wikipedia can be very helpful: if you are lucky, the article you are reading has been edited thoroughly by experts on the topic, and then a Wikipedia article can be more helpful than a printed publication. However, on Wikipedia you are not always lucky, and moreover, information on Wikipedia can change any time. Always look for printed, stable sources that confirm what you found on Wikipedia.

Try to use sources that can at least be ordered on-line, and are written in a language most readers understand.

Publication details include:

- names of the authors;
- title;
- publisher, journal, or institute (depending on the type of source)
- volume and number (if the publication is part of a series)
- article number or page numbers (when referring to a book, indicate on which pages or in which section the reader can find the information for which you refer to the book)
- date (for a website, give the exact date on which you studied the website; for other publications the year suffices)

1.10 Structure

1.10.1 Sections and subsections

Make sure your report is divided into sections and subsections in a way that helps the reader understand where to find what in your report. Every section and subsection should have a clear topic that sets it apart from the other sections, or from the other subsections in the same section, respectively. Try to give similar parts of the report similar structure: for example, if you describe two complicated algorithms, and the description of one algorithm is divided into three subsections (approach, implementation, and analysis), then you should use the same subdivision into subsections for the other algorithm.

Give each section or subsection a concise title that describes the contents of section. Do not make the titles too concise. Suppose your document describes, for example, an incremental algorithm and a divide-and-conquer algorithm, each in its own section with subsections for approach, implementation and running time analysis. Then a subsection title "Running time of the divide-and-conquer algorithm" is more useful than a subsection title that just reads: "Analysis".

Start each section with a paragraph that gives an overview over the whole section, rather than immediately diving into the first subsection.

Avoid having too many small subsections, and avoid too large subsections: at the lowest level of subdivision, a ((sub)sub)section should be at most a few pages.

1.10.2 Paragraphs

Divide the text within each section, subsection or subsubsection into meaningful paragraphs. Typically a paragraph starts with a sentence that defines the topic of the paragraph. The rest of the paragraph then develops that topic. In paragraphs in which you give arguments to support a certain statement, this usually builds up to the last sentence in which you conclude that the statement is true.

The first sentence of a paragraph should also clarify how the paragraph relates to the previous paragraph: use appropriate linking words or phrases such as: "however", "moreover", "nevertheless", "as a result", "this is caused by", "in particular", "in conclusion", "to prove/compute/ solve/test this", "next" etc. Avoid linking paragraphs together by writing things such as "Another thing we should mention is ...": this does not help the reader to understand the structure of your text. If the paragraph is part of a list, for example a series of three paragraphs each discussing a particular problem with a certain algorithm, then you should introduce that list with a sentence that explains that there are three problems, and start the paragraphs with words such as "firstly", "secondly", and "lastly".

Paragraphs should not be too long. Paragraphs longer than 1/4 of a page should definitely be broken up into smaller paragraphs. Paragraphs should not be too short either. It is possible that the structure of the text results in some very short paragraphs, but if you have several paragraphs in a row that each consist of only one or two sentences, then the paragraph structure will not help the reader much, and you should probably join some of these paragraphs into one paragraph.

2 Lay-out and typography

2.1 (No) Cover page

With larger reports, it may be good to have a separate cover page, to include a table of contents (and maybe even a list of tables and figures) and to make sure that every chapter starts at the top of an odd-numbered page. However, in a report of only a few dozen pages this would be a waste of paper: a cover page, a table of contents and white pages are not necessary and not appreciated.

2.2 Titles of the document, sections, subsections and paragraphs

For the typography of titles of sections, tables, and the report as a whole there are many options: you can make use of different fonts, different font sizes, upper- and lowercase letters, and white space below and above the titles. My advice is to stick to the LaTeX defaults: titles are typeset in boldface; higher-level titles are typeset in bigger fonts with more white space around. Use a capital for the first letter of the first word of a title.[2]

2.3 Text

If you want to emphasize certain words or phrases in your text, use italics (boldface and underlining is considered over the top); in LaTeX you can do this with the command `\emph`. When you define a concept, emphasize it (only the first time):

> A *spanning tree* of a graph G is a tree whose vertices are the vertices of G, and whose edges are a subset of the edges of G. A *minimum spanning tree* of a weighted graph G is a spanning tree of G that has minimum total edge weight among all spanning trees of G.

2.4 Bibliography

Use a consistent lay-out for the bibliography. Format all sources of the same type in the same way. It is common practice to put the title of the whole publication in italics. Thus, if you refer to an article in a journal or an abstract in the proceedings of a conference, it is the title of the journal or the proceedings (conference) that is in italic font, while the title of the paper itself is in normal font. Here are examples of how you could typeset references to a journal paper, a conference paper, and a PhD thesis, respectively:

- Herman Haverkort and Freek van Walderveen: Locality and bounding-box quality of two-dimensional space-filling curves, *Computational Geometry (CGTA)*, 43(2):131–147 (2010).
- Mark de Berg, Otfried Cheong, Herman Haverkort, Jung-Gun Lim, and Laura Toma: I/O-efficient flow modeling on fat terrains, *Proc. 10th Int. Workshop on Algorithms and Data Structures (WADS)*, 2007, LNCS 4619:239-250.
- Micha Streppel: *Multifunctional geometric data structures*, PhD thesis Eindhoven University of Technology, Dept. of Computer Science, 2007.

2. Different publishers and style guides give different rules for which words in a title should start with a capital. Some publishers and style guides prefer to capitalize only the first word of the title. Another common style is to use capitals for the first letters of all “important” words in the title. Unfortunately this raises many questions about which words get a capital and which words do not. To stay on the safe side, only capitalizing the first word is a good choice.

3 Language

In this section you will find some advice on how to write your report in proper English, in the proper style, and how to avoid common pitfalls in mathematical writing.

3.1 Style: personal versus impersonal

3.1.1 You becomes we

In technical writing, we avoid the second person ("you", "your"). To address the reader, use "we". In this case "we" can be understood as: the reader(s) and I/we, the author(s).

> In Section 2, we have seen how to ...
> ~~In Section 2, you have seen how to ...~~

3.1.2 I becomes we

The singular first person ("I", "me", "my") is also avoided. If a text has multiple authors, they are all responsible for the complete text, and write as if they did everything together:

> We have tested our algorithm on ...
> ~~I have tested my algorithm on ...~~

3.1.3 It becomes we

Do not overuse the passive form: it can make your text look clumsy.

> ~~First, a heap is built from the input elements. Then a loop is executed: in each iteration, an element is extracted from the heap, the element is appended to the sorted list, and the heap property is restored by calling MinHeapify. This is repeated until the heap is empty.~~
> The algorithm first builds a heap from the input elements. Then we execute a loop: in each iteration, we extract an element from the heap, append the element to the sorted list, and call MinHeapify to restore the heap property. We repeat this until the heap is empty.

In my opinion, the second description is much more pleasant to read. I believe the use of "the algorithm", and then even "we", invites the reader to identify himself with the computer that is running the algorithm: the reader is invited to be in the middle of the action and to imagine what would happen if he would perform the steps of the algorithm himself.

3.1.4 Her, him, and it

In English, "him" and "her" are only used to refer to persons. Objects (and, usually, even animals) are referred to by "it". However, when using "it" in mathematical writing, it is important that you make absolutely sure that it is absolutely clear (even for the slightly careless reader) what "it" refers to. Often it is better to give the object that is being referred to a name:

> ~~If the vertex extracted from the queue has the same label as the starting vertex, he is coloured blue and his neighbours are enqueued.~~
> If the vertex v extracted from the queue has the same label as the starting vertex, then we colour v blue and enqueue its neighbours.

In case you need to refer to single third persons, an easy and non-obtrusive way to do this in a gender-neutral way is to sometimes use “she” and “her”, and sometimes use “he”, “him” and “his”—but keep it locally consistent, as people do not usually change gender from one sentence to another. Another non-obtrusive solution is to use the plural. This is not always possible, but often it is.

3.2 Style: variation versus monotonicity

In literary writing, repeating words and phrases in consecutive sentences is sometimes considered evidence of poor style or a poorly-educated author’s small vocabulary. However, in mathematical writing, precision is everything. When referring to the same concept, use the same word: avoid synonyms, so that the reader does not have to worry about any subtle differences between the definitions of the two words:

> ~~If u is a vertex of G, then we colour u blue, whereas if u is a node of H, we colour u red.~~
> If u is a vertex of G, then we colour u blue, whereas if u is a vertex of H, we colour u red.

When two things are analogous, then use analogous wording to describe them:

> ~~If u is a vertex of G, then we colour u blue, whereas we colour u red, if u is a vertex of H.~~
> If u is a vertex of G, then we colour u blue, whereas if u is a vertex of H, we colour u red.

3.3 Grammar and structure of sentences

3.3.1 Length of sentences

Make sentences not too short and not too long. Occasionally a long sentence is hard to avoid, but as a rule, three lines is too long.

3.3.2 Much, many, amount, and number

The words “many” and “number” are used with the plural forms of countable concepts. The words “much” and “amount” are used with uncountable concepts, which do not have plural forms:

> ~~many memory, many time, many money, the number of memory, the number of time, the number of money~~
> much memory, much time, much money, the amount of memory, the amount of time, the amount of money
> ~~much vertices, much points, the amount of vertices, the amount of points~~
> many vertices, many points, the number of vertices, the number of points

3.3.3 Then and than

In comparisons, use “than”, not “then”:

> ~~The number of edges of G is greater then the number of edges of H~~
> The number of edges of G is greater than the number of edges of H

3.3.4 To be done and to have been done

When describing algorithms, it is important to be clear about what has been done, what is being done, and what still remains to be done. Note that the Dutch “is berekend” does not translate to “is computed”:

Dutch	English	
is berekend	has been computed	(computation is complete and results are available)
wordt berekend	is computed	(computation is ongoing and results are still to come)

3.3.5 References

A common mistakes is to use references to the bibliography as words. This cannot be done:

> ~~In [4] it is proven that...~~
> De Berg et al. [4] prove that...

Note how the second sentence is also more informative to the reader, making it much easier for him to guess or remember which publication is meant by reference [4].

3.3.6 Type consistency

Avoid type mismatches, like you would when programming. Often the result of a type mismatch is that it is unclear what the author means. Here are some examples:

Wrong: *“We assume the set of points S does not have the same x-coordinate.”*
Problem: a set does not have an x-coordinate.
Correct: *“We assume that no two points from S have the same x-coordinate.”*

Wrong: *“If the weight of edge e is more than f, then we change f to e.”*
Problem: is f a weight or an edge?
Correct: *“If the weight of edge e is more than the weight of f, then we change the weight of f to the weight of e.”* Or: *“If* weight(e) > weight(f)*, then we change* weight(f) *to* weight(e)*.”*

Wrong: *“The algorithm runs in $O(n)$.”*
Problem: $O(n)$ of what? time? memory? something else?
Correct: *“The algorithm runs in $O(n)$ time.”* Or: *“The algorithm runs in $O(n)$ memory.”*

Wrong: *“This algorithm is the easiest to solve.”*
Problem: an algorithm is not something that can be solved; an algorithm is something that is used to solve a problem.
Correct: *“This problem is the easiest to solve.”* Or: *“This algorithm is the easiest solution.”*

3.4 Words and concepts to be careful with

3.4.1 Miljard, billion etc.

Unfortunately, the English-speaking world got confused about the meaning of the word “billion”. It seems that by now, in English this is practically always taken to mean 10^9. However, the non-English speaking part of Europe still believes that a billion is 10^{12}. To avoid confusion in an international context, do not use words for any numbers greater than 999 999 999.

3.4.2 Maximum and maximal

In mathematical writing, there is a subtle difference between “maximum” and “maximal”. “Maximum” means: as large as possible. “Maximal” means: cannot be enlarged. Consider, for example, *independent sets* in graphs: a subset S of the nodes of a graph $\mathcal{G} = (V, E)$ is independent if and only if there is no pair of nodes $u, v \in S$ that is connected by an edge $(u, v) \in E$. A *maximum independent set* is an independent set I such that no independent set is larger than I. A *maximal independent set* is an independent set I such that, if you would add any vertex of $V \setminus I$ to I, then I would not be an independent set anymore.

3.4.3 Worst-case complexity

Be precise when making statements about the asymptotic running times of algorithms. The following five statements all mean something different:

- “The algorithm needs $O(n^2)$ operations.” This statement gives an upper bound on the number of operations in the worst case (and thus, also in the best case). It is possible that the bound is not tight and the algorithm is actually faster, even in the worst case.
- “The algorithm needs $\Omega(n^2)$ operations in the worst case.” This statement gives a lower bound on the number of operations in the worst case. In the best case, the algorithm may be faster.
- “The algorithm needs $\Omega(n^2)$ operations.” This statement gives a lower bound on the number of operations in the *best* case (and thus, also in the worst case).
- “The algorithm needs $\Theta(n^2)$ operations in the worst case.” This statement says that for any n that is large enough, there are inputs for which the algorithm actually needs a quadratic number of operations (never more). However, there may also be inputs on which the algorithm is faster.
- “The algorithm needs $\Theta(n^2)$ operations.” This statement says that the algorithm *always* uses a quadratic number of operations when n is large enough, even in the best case.

Note that, for example, if one algorithm uses $O(n^3)$ operations, and another algorithm uses $O(n^2)$ operations, there is no way to know which algorithm is faster, not even for large n. Big O’s only give *upper* bounds on the running time—without *lower* bounds (Θ- or Ω-bounds), it is very well possible that, for example, both algorithms actually run in $O(n \log n)$ time.

3.4.4 Average-case complexity

Do not mention it. Analyzing the average-case running time of an algorithm is only conceivable if you know the probability distribution of all possible inputs. This is almost never the case.

3.4.5 Best-case complexity

Do not mention it.

3.4.6 Expected running time

“Expected running time” is a technical term which means: the average running time over a given (possibly infinite) set of runs of a randomized algorithm *on the same input*; note that this input may be a worst-case input.

When explaining the workings of your algorithm, it is perfectly fine to give your intuition about what you expect the running time of your algorithm to be in practice. However, do not use the exact phrase “expected running time” then. Do not make any statements about the “expected running time” of an algorithm, unless you can do the necessary (and usually very difficult) math to back it up.

3.4.7 It is easy to see

Never write “It is easy to see that...”. If it is, it is also easy to explain, and you should just do that. If you do not know how to explain it, then, apparently, it is not easy, and claiming that it is easy will make the reader either suspicious or insecure.

3.4.8 Respectively

“Respectively” is used for the same purpose as “respectievelijk” in Dutch, but in English, it is put in a different place in the sentence:

> ~~The blue and the red points lie respectively inside and outside the rectangle.~~
> The blue and the red points lie inside and outside the rectangle, respectively.

Often it is clearer to avoid using “respectively” altogether:

> The blue points lie inside the rectangle whereas the red points lie outside the rectangle.

3.5 Spelling

There are some things your spell checker may not catch: note the difference between “to prove” (the verb “bewijzen”) and “the proof” (the noun “het bewijs”).

Do not use “thru” and “til(l)”: these are informal short forms of “through” and “until”, respectively.

Note the following irregular plurals: leaf–leaves, vertex–vertices, index–indices. However, when using “index” in the meaning of “data structure”, then the plural is simply “indexes”.

Avoid abbreviations. Abbreviations that you defined yourself are always much harder to remember for the reader than they are for the authors. Also avoid abbreviations such as “i.e.”, “e.g.”, “w.l.o.g.”, “iff”: many readers (and many authors) do not know *exactly* what they mean. To communicate most clearly, it is much better to write the words out completely.

3.6 Numbers

In some countries, the number three thousand two hundred one is usually written as “3.201” and the number three and two hundred one thousandths is written as “3,201”, while in other countries it is exactly the other way around. To reduce the risk of confusion, use narrow spaces (\, in LaTeX) to separate the thousands, and use a point to separate the fractional part from the integral part:

> ~~3.201 and 3,201~~
> ~~3,201 and 3.201~~
> 3 201 and 3.201

Use words for quantities up to twelve:

> ~~This graph contains at most 8 nodes.~~
> This graph contains at most eight nodes.

3.7 Punctuation

3.7.1 Commas (,)

As a rule of thumb, use commas wherever a pause would help the reader in parsing the sentence correctly. In case of doubt, insert a comma: it is better to have too many commas than to have too few.

In particular, commas are often used around words or expressions that interrupt the normal flow of a sentence:

> This is, of course, not straightforward to implement.

Commas are also used at the end of subordinate clauses, especially if they are long:

> If all children of u are coloured red, we colour u blue.

Relative clauses start with a comma if the clause gives identifying information about its antecedent, but not if the clause only gives additional information that does not contribute to identifying the antecedent. The difference can be quite subtle. For example, consider three dots in a row: the leftmost dot and the dot in the middle are blue, and we do not know the colour of the rightmost dot yet. Now, if we would write: "u is the rightmost dot, which is blue" (with a comma), then the relative clause "which is blue" does not contribute to identifying u, so the reader must assume that u is simply the rightmost dot, and in addition, we inform the reader that the rightmost dot is blue. However, if we would write: "u is the rightmost dot which is blue" (without a comma), then the relative clause "which is blue" is part of the definition of u. Then there are two possibilities: either the rightmost dot is blue, and then u is the rightmost dot, or the rightmost dot is not blue, and then u is the middle dot.

Commas are used to separate items in a list, but usually not if the list contains only two simple items:

> The data structure stores vertices, edges, and faces.
> The data structure stores the vertices of degree at least five that have a neighbour of degree one, and the edges between those vertices.
> ~~The data structure stores vertices, and edges.~~
> The data structure stores vertices and edges.

If the items in a list are each on a separate line, or if the items are grammatically complex (in particular if they contain commas themselves), then semicolons can be used instead of commas.

Commas cannot be used between grammatically complete sentences. There one should use a full stop, a colon, or a semicolon, according to the rules explained below.

3.7.2 Colons (:) versus commas

Colons are used to indicate that what follows gives an explanation or more details of what came before. Commas cannot be used in this way:

> ~~Take the introduction very seriously, this should be the best-written part of the whole report.~~
> Take the introduction very seriously: this should be the best-written part of the whole report.

3.7.3 Semicolons (;) versus commas

Semicolons provide a way to separate expressions that is stronger than a comma, but less strong than a full stop.

Semicolons are frequently used instead of commas to separate items in a list, if the items are grammatically complex (especially if they contain commas) or if they are on separate lines.

3.7.4 Apostrophes (')

Apostrophes are used to make possessive forms of nouns:

the point's coordinates	meaning: the coordinates of the point
the points' coordinates	meaning: the coordinates of the point**s**

In formal writing, do not use apostrophes to make contractions:

~~it's, we'll, we've, we're, don't, doesn't, can't, won't, isn't~~
it is, it has, we will, we have, we are, do not, does not, cannot, will not, is not

Never write "it's" in formal writing: the neutral possessive pronoun ("zijn") is "its" (without apostrophe); "it's" means "it is" or "it has" and should be written out completely:

~~It's impossible to compute the original key from it's hash value~~
It is impossible to compute the original key from its hash value.

3.7.5 Hyphens

When combining words in English, there are no clear rules about when to keep the words separated by spaces ("data base"), when to combine them with a dash ("data-base"), and when to write them as one word ("database"). In case of doubt, use spaces, or use Google to see how the rest of the world writes it. However, in some cases, hyphens are necessary.

When combining words with spaces, the combinations are right-associative. This can be changed with hyphens: a hyphen has the effect of putting parentheses around the words connected by the hyphen. Compare:

spelling	with parentheses	meaning
worst case analysis	(worst (case analysis))	a case analysis that is really badly done, for example with lots of mistakes
worst-case analysis	((worst case) analysis)	an analysis of the worst case

3.8 Typesetting formulas

Use a consistent typography for mathematical symbols: use italics for variables (in LaTeX: use the math environment):

~~x-axis~~
x-axis

Make sure that symbols in figures are typeset in the same way as in the main text (consider to use IPE: this is a figure editor that uses LaTeX for typesetting).

Put names of standard functions in normal font. You can also define your own "standard" functions. If LaTeX has a command for your function, use it (for example, `\sin`), otherwise you can use `\mathrm` to put text in normal font:

~~$sin(x + width(y))$~~
$\sin(x + \mathrm{width}(y))$

For variable names of more than one letter, you can use `\mathit`. Without `\mathit` there will be too much space between the letters. Do not use dashes in variable names, because these can be confused with minus signs:

> ~~$Cluster - Quality$~~
> $\mathit{ClusterQuality}$

Use double-stroke letters for the standard sets of numbers (in LaTeX, use `\mathbb`):

> $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$

3.9 Mixing mathematical symbols and text

Do not start a sentence with a mathematical symbol, and separate symbols in different formulas by words:

> ~~$f(x)$ is positive for all $x \geq 1$, $g(x)$ is negative for all $x \geq 0$, and $h(x)$ is at least 0 and at most 1 for all $x \geq 0$.~~
> The function $f(x)$ is positive for all $x \geq 1$, the function $g(x)$ is negative for all $x \geq 0$, and $h(x)$ is at least 0 and at most 1 for all $x \geq 0$.

4 Pseudocode

Explaining an algorithm in paragraphs of normal English text is difficult. Therefore it is often useful to include a description in pseudocode as well—as well, that is, *not instead of* an intuitive textual description. The main goal of using pseudocode is to clarify the flow of control (loops and branching) in an algorithm. Pseudocode should be readable by anybody who understands basic algorithmic concepts such as loops, variables, and function calls. In particular, it should be readable by anybody who understands the basic concepts but does *not* know C++, Java, Pascal, or whatever your favourite programming language is. Remember that pseudocode is used to explain how an algorithm works; not to explain how to code it.

For typesetting pseudocode, there are several LaTeX packages available, for example `algorithm2e`.

4.1 Loops

You can use the following structures for loops:

- A for loop in which *variable* starts at *firstValue*, is increased by one after each iteration, and which terminates as soon as $variable > lastValue$:

```
for variable = firstValue to lastValue do
    statement
    ⋮
```

- A for loop in which *variable* starts at *firstValue*, is decreased by one after each iteration, and which terminates as soon as $variable < lastValue$:

```
for variable = firstValue down to lastValue do
    statement
    ⋮
```

- A for loop in which *variable* takes on the values of the elements of a sequence (such as a linked list), in order:

```
for each variable in sequence do
    statement
    ⋮
```

- A while loop:

```
while condition do
    statement
    ⋮
```

- A repeat-until loop:

```
repeat
    statement
    ⋮
until condition
```

There are many other formats for loops in pseudocode that are equally acceptable. However, do not use Java/C-style for-loops (**for** (...; ...; ...)) and switch-statements in pseudocode: they are not universally understood. Avoid using a while-statement when the same loop can be written as a for-loop.

4.2 Branching

You can use the following structure for conditional statements:

```
**if** $condition$ **then**
    $statement$
    $\vdots$
**else if** $condition$ **then**
    $statement$
    $\vdots$
$\vdots$
**else** $\triangleright$ $condition$
    $statement$
    $\vdots$
```

The **else** blocks are optional. For comprehensibility it often helps to add a comment after **else**, mentioning conditions that hold whenever the **else** clause takes effect. For example, if x can have two values, **true** or **false**, and actions should be taken depending on the value of x, one could write an **if** statement like this:

```
**if** $x =$ **true then**
    $statement$
    $\vdots$
**else** $\triangleright$ $x =$ **false**
    $statement$
    $\vdots$
```

4.3 Operators

Do not use programming-language-specific or other non-standard operators such as `++`, `--`, `!`, `!=`, `^`, `*`, $\uparrow$, $\downarrow$. Instead, use words or mathematical symbols: "increment", "decrement", "not" or $\neg$, "not equal to" or $\neq$, "XOR" or superscript (depending on the intended meaning), $\cdot$ or superscript (depending on the intended meaning), max, min.

For the assignment operator, you can use either $\leftarrow$ or $=$ (but be consistent). It is possible to assign tuples of values to tuples of variables; if you do, use parentheses:

```
    ~~$i, j \leftarrow j^2, i$~~
    $(i, j) \leftarrow (j^2, i)$
```

4.4 Variables and arrays

Choose descriptive variable names: using i for a counter in a simple loop is fine, but otherwise, use multi-letter variable names. All variables should be declared, either as an argument to the function call, or by initializing them.

By default, if A is an array of n elements, then the first element of A is $A[1]$, and the last element is $A[n]$. Sometimes it is easier to use another range, for example using $A[0]$ as the first element and $A[n-1]$ as the last element. If you do, make sure there is no confusion and declare it clearly, for example: "Let $A[0...(n-1)]$ be an array of n elements."

By default, all variables are local; a change to the value of a variable is not visible to the calling function. Function parameters are passed by value: the function gets a local copy of the parameter and any changes to it are not visible to the calling function. Arrays and other data structures storing multiple values are passed by reference. For example, consider the following four algorithms:

```
Algorithm UselessA($A$):
$A[1] \leftarrow 1$
$x \leftarrow 2$
$y \leftarrow 3$
```

```
Algorithm UselessB($A, x, y$):
$A[1] \leftarrow 1$
$x \leftarrow 2$
$y \leftarrow 3$
```

```
Algorithm UselessC($A$):
$A[1] \leftarrow 1$
return $(2, 3)$
```

```
Algorithm Test:
Let $A$ be an array
$A[1] \leftarrow 0$; $x \leftarrow 0$; $y \leftarrow 0$
UselessA($A$)
print $A[1], x, y$
UselessB($A, x, y$)
print $A[1], x, y$
$(x, y) \leftarrow$ UselessC($A$)
print $A[1], x, y$
```

Algorithm TEST would print 1, 0, 0, 1, 0, 0, 1, 2, 3. Note how USELESSA and USELESSB fail to change the values of x and y, whereas assigning the return value of USELESSC to (x, y) does the trick.

4.5 Function calls

Do not call functions which you did not define. For operations on data structures, rather use a description in English that makes clear what the operation is supposed to do. Note how in the example below, the first two attempts fail to make clear what is appended to what:

```
~~Append($L, M$)~~
~~$L$.append($M$)~~
Append $L$ to $M$
```

4.6 Comments

Use comments to clarify the pseudo-code where useful. Comments can be marked by starting them with ▷ and/or using *italic font*. Particularly useful comments are:

- comments that explain the purpose of a variable (the meaning a variable will have during execution of the code);
- comments that state a condition that will help understanding the code that follows (see Section 4.2 for an example);
- comments that explain what the following statements in the code will achieve.

4.7 Length and integration in the text

Pseudocode should never be broken at the bottom of a page. Make sure that the text of your document refers to the pseudocode. Note that the purpose of pseudocode is to clarify the main text, not to replace it. The main text should still make sense if the reader decides to skip the pseudocode.

5 Charts

To get a quick overview over your data, and to give the reader of your report a quick overview over your data, it can be very useful to make charts (graphical presentations) of your measurements. While you are still doing your research, automated tools to make charts are the way to go. When you are writing your report, you could take some time to draw a well-designed chart by "hand". Carefully consider the design of your charts: the way in which you represent the data determines what structure in the data catches the eye and what structure remains hidden. To illustrate this, Figure 1 shows three charts of the same data.

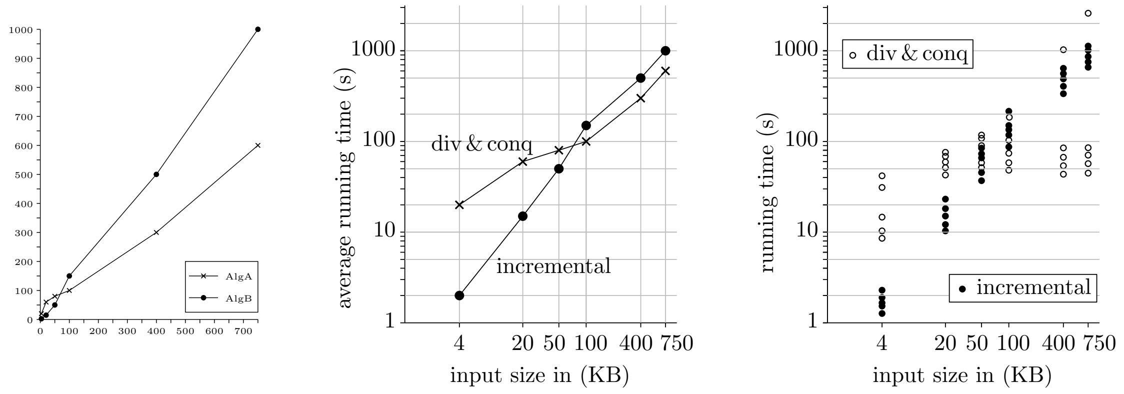

Figure 1: Three graphical presentations of the same data

In the chart on the left, it is unclear what type of numbers are shown. It lacks labels on the axes, indicating what is shown (input size and average running time), and what the units of measurement are (kilobytes and seconds, respectively). The values shown are unclear, because the font is just too small. In the charts in the middle and on the right, this is improved by using larger font, and, on the horizontal axis, by printing the values of the data rather than the values of regularly spaced grid points.

All charts show the results of experiments with two algorithms: an incremental algorithm and a divide-and-conquer algorithm. In the chart on the left, it is hard to see which results are from which algorithm. This is due to two problems. Firstly, the labels of the lines in the chart on the left are in a separate legend, in which the top-to-bottom order of the lines is, inconveniently, the *opposite* of the general top-to-bottom order in the data. In contrast, in the other two charts the labels are placed close to the data. Secondly, in the chart on the left the labels are unclear: for the reader, it will be much more difficult to remember which algorithm is meant by "AlgA", than to remember which algorithm is meant by "incremental".

Finally, note that the three charts emphasize very different aspects of the data. The chart on the left focusses on how, on average, the divide-and-conquer algorithm is faster than the incremental algorithm if the input is large. It is hardly visible that for small inputs, the incremental algorithm is *much* faster. The chart in the middle uses logarithmic scales[3] to create a more balanced picture, which shows the results for small inputs as clearly as the results for large inputs. Both in the chart on the left and in the chart in the middle, much information is lost because we only show the average running times. You could also consider plotting the results of each individual experiment, as in the chart on the right. This reveals, for example, that usually the running

3. When using a logarithmic scale, a constant distance along the scale corresponds to a difference in value by a constant factor. When we put values between x_0 and x_1 on a scale of length d, the value x is positioned at distance $d \cdot \log(x/x_0)/\log(x_1/x_0)$ from the lower end of the scale.

time of the divide-and-conquer algorithm in this experiment hardly depends on the input size, but sometimes it is much slower than the other algorithm.

Scale the charts in your report to the right size: do not waste a full page if, with a bit of care, an equally informative and equally clear chart could be drawn that is much smaller. Do not try to save space by letting a linear vertical axis start at any other value than zero: this would make the chart really confusing.

It is allowed to use colours for clarity, but do not rely on them: readers and reviewers may print your report in black-on-white.

Just like with pseudocode and figures, always refer to the chart from the main text of the document. A chart that is never referred to is easily overlooked by the reader, because the reader will usually not miss it.

6 Tables

When putting numbers in tables, make sure that it is clear what the numbers mean. What is the unit of measurement? (for example: milliseconds, seconds, or minutes?). Label the table so that it is clear what data is shown in the table.

Do not give more digits than necessary: often everything after the second digit is uninteresting because these digits are unreliable (the result of random fluctuations in the tests) and irrelevant for comparisons.

Make sure that it is easy to see which table entries are in the same row. There are several tricks you can use to facilitate this, for example:

- use proper spacing between rows;
- add extra spacing or a horizontal line after every third, fourth or fifth row;
- make the columns not wider than necessary;
- use alignment on the right for the contents of the leftmost column;

Here are two versions of the same table, illustrating the advice given above.

data set	algorithm A	algorithm B
Friesland	19.455	8.256
Groningen en Drenthe	23.415	9.103
Overijssel	26.524	8.992
Gelderland	40.157	35.227
Limburg	37.849	35.306
Noord-Brabant	35.025	32.008
Zeeland en Zuid-Holland	51.087	57.938
Noord-Holland en Flevoland	32.308	15.707
Utrecht	18.823	7.215

Table 1: average running time (s) of 10 runs

data set	*alg. A*	*alg. B*
Friesland	19	8
Groningen en Drenthe	23	9
Overijssel	27	9
Gelderland	40	35
Limburg	38	35
Noord-Brabant	35	32
Zeeland en Zuid-Holland	51	58
Noord-Holland en Flevoland	32	16
Utrecht	19	7

Just like with pseudocode, figures, and charts, always refer to the table from the main text of the document. A table that is never referred to is easily overlooked by the reader.