

Chapter 11

Algorithm Description

11.1 Goals—multiple audiences, graphics, sophisticated audiences

This assignment practices

- writing to a technically sophisticated audience,
- writing to two different audiences at the same time,
- describing a recursive algorithm, and
- proper use of displays and figures.

11.2 Audience assessment—writing for multiple audiences

The Dinky Database Dantai¹ has had complaints from their customers about the slowness of the zip code sorting operation. You have been hired as a consultant to diagnose the problem. They can't afford your consulting fees for the time it would take you to code and debug the routines yourself, so you have to prepare a report to their programmers explaining the fixes to be done. You cannot talk with the programmers who wrote the code, because they are all working for Bit Bucket Enterprises now. The new DDD programmers are mainly entry-level programmers (a euphemism for high school and junior college students working in their spare time).

On examining the code, you find that

- zip codes are stored alphabetically as variable-length character strings,
- all sorting is internal (all zip codes read into memory before sorting),
- the sorting is done by a variation of bubble sort (an order n^2 algorithm),
- comparisons are done by passing the pointers to the two strings to a function, and
- exchanges are done by swapping two pointers in an array, not by copying entire strings.

Because Canadian and other foreign addresses use non-numeric mail routing codes, and even US codes vary in length, you decide that the data representation is appropriate, and that alphabetical order is the correct ordering for zip codes (00002-4455 is very close to 00002, not to 24455). The comparison and exchange operations seem to be correctly written, with only minor gains in efficiency possible from re-implementing them more carefully.

After checking with DDD, you find that customers have not complained about the number of entries they can sort, only about how long it takes. This means that internal sorting is a reasonable choice, and the extra complexity and speed penalty for external sorting is not justifiable.

You cannot find any reason for using bubble sort, and decide to recommend a faster sorting technique (quicksort, heapsort, merge sort, shell sort, radix sort, . . .). You have to explain briefly why you are recommending a change of

¹*Dantai* is the Japanese word for a private corporation.

algorithms, then give a detailed description that the programmers can understand. You do not have to give a full theoretical analysis of the algorithm, but a few sentences to explain why you chose the algorithm you did would be helpful.

Your report will be read by two audiences: the managers of DDD and the programmers. The managers have no formal training in computer science, but want to know that you've solved their problem. The programmers know a little computer science, but are not familiar with sorting algorithms. The programmers may know as much as you do, but because they can't write nice reports, they get sweatshop wages while you earn consultant rates.

11.3 Writing process—algorithm description

Your first task is to choose an algorithm. You may already know one that is good enough. If not, Knuth's *Volume 3—Sorting and Searching* [Knu73] should provide the information you need. Of course, you don't want to read all of Volume 3 before making up your mind. If you don't have a copy of Knuth, almost any sophomore or junior data structures book will describe a few of the more important sorting algorithms. You may want to check the serial published by the ACM containing nothing but algorithms for various problems [ACM].

Choose your sorting algorithm to be fast and to require minimal changes to the data structures in the existing program. You know that they use bubble sort, and so the information is stored in an array, with comparison and exchange operations available. If your algorithm requires some other data structure, you will have to describe how to build that data structure as well. For example, merge sort may be somewhat easier to describe than quicksort, and less subject to “off-by-one” errors, but the simplest implementation requires linked lists, which would increase the memory requirements. If you do describe merge sort, be sure to include the appropriate information about error-checking and pointer manipulation: in the past, many students have used `x->next->next` in their pseudo-code without warnings about odd list lengths and checking for null pointers.

When describing an algorithm, you have to pay careful attention to the level of abstraction. You do not want to describe everything in gory detail—that's the programmer's job. You also don't want to treat the whole problem as a single black box either. If you knew the programmers were highly skilled, it might be enough to tell them “Use quicksort!”, but for this crew you'll need more detail than that.

The main parts of an algorithm description are

- purpose (what is the result of running the algorithm),
- data structure (what is manipulated by the algorithm),
- technique (what steps does the algorithm perform),
- justification (proof of correctness, often reduced to hand-waving),
- analysis (speed, space, cost, ...).

Different audiences and different purposes demand attention to different aspects of the algorithm description. When presenting a new algorithm for a well-known problem to the research community, the emphasis is usually on technique, justification, and analysis. In that situation, you're trying to show how much better you've done than previous researchers. When presenting a new data structure (say a better representation of a priority queue), the data structure and analysis might be stressed.

For this report, you will be presenting a well-known algorithm as the solution to a specific problem. Here data structure and technique are most important. You can point to the literature for a proof of correctness and the details of the analysis.

Some of the tools used for proving correctness are also useful for explaining the technique, so don't ignore them entirely. For example, the termination condition for a loop needs to be known both for the justification and for the coding. Loop invariants (statements that describe a property that is unchanged as the loop is repeated) are essential for correctness proofs, and handy for generating the loop in the first place. David Gries gives a detailed presentation on using loop invariants to help write programs [Gri83].

11.4 Figures and displays

A manager reading your report will look at the executive summary and at the figures, to see what the report is about and whether it is worth passing on to the programmers who'll read the complete report. It should be possible to get a fair idea of the content of the report just from looking at the figures and reading the captions. Look at the articles in *Scientific American* for an excellent example of how figure captions can be used to convey most of the substance of a technical article.

You have several opportunities for figures in this report—you can use them for the pseudo-code for the algorithm, for showing the change in the data structure as the sorting progresses, for displaying the relative speed of different algorithms as a function of the number of items to sort, and so forth.

There are two ways to insert non-textual material into a report: *figures* and *displays*.

A figure can appear anywhere in the report, and usually appears on the same page as, or slightly after, the first reference to it. Figures are numbered and have a caption explaining what the figure is illustrating. Each figure must be referred to somewhere in the text, either as part of a sentence, like this: “Figure 37 illustrates the relationship between . . .”; or as a parenthetical remark, like this: “(See Figure 38.)”. Remember to capitalize the word “figure” when it is used as part of the name for a particular figure.

A display differs from a figure in that it is inserted in a fixed place, and is not allowed to “float” to a different place in the text. Displays are used for math formulas, very small program fragments, and quotations that are too long just to put in quotes in the middle of a paragraph. Some empty space is usually inserted before and after each display to set it off from the surrounding text, and displays are often centered on the page. Mathematical formulæ are usually inserted as displays, and are grammatically part of the sentence before them. For example, I can define the golden ratio as the positive solution to the quadratic equation

$$x^2 = x + 1 ,$$

and continue writing the same sentence.

11.4.1 Graphics

Pictures are often the clearest way to explain data structures, particularly when pointers are required. Whenever a picture is used, it should be a numbered, captioned figure. Figures should be (nearly) comprehensible without the accompanying text. Make sure the caption explains what the figure means. Something like “Figure 3. Step 2 of the algorithm.” is nearly useless. A better caption might be “Figure 3. Swapping out-of-order data elements pointed to by i and j.”

The pictures you are likely to draw are block diagrams, symbolic representation of pointer structures, maps of contiguous sections of memory, and graphs showing how something (time, cost, current, voltage, . . .) varies when changing various parameters. The rest of this section will talk only about numerical graphs, though you probably will need to generate other sorts of graphics, even for this assignment.

Proper treatment of graphics is really beyond the scope of this class. A fairly good introduction is included in Chapters 8 and 9 of Huckin and Olsen [HO91, 137–184]. If you want a comprehensive treatment of presenting numerical (mainly statistical) data in a professional form, the best book is Tufte’s *The Visual Display of Quantitative Information* [Tuf83] on reserve in the Science Library. Other books can be found under subject headings like “Engineering Graphics,” “Graphic Methods,” and “Computer Graphics.” For examples of how not to present numerical information, see almost any mass-market magazine or newspaper. *USA Today* is particularly good at distorting tiny data sets into fancy pictures that hide the data. Another particularly good collection of bad examples can be found in a book by *Time Magazine* illustrator Nigel Holmes [Hol84], although they are presented as if they were the best way to illustrate data.

Tufte summarizes his theory of data graphics as follows [Tuf83, 105]:

Five principles in the theory of data graphics produce substantial changes in graphical design. The principles apply to many graphics and yield a series of design options through cycles of graphical revision and editing.

- Above all else, show the data.
- Maximize the data-ink ratio.

- Erase non-data ink.
- Erase redundant data ink.
- Revise and edit.

Tufte also has some strong comments about “chartjunk”, the cluttering of graphs with non-informative decorations [Tuf83, 121]:

Chartjunk does not achieve the goals of its propagators. The overwhelming fact of data graphics is that they stand or fall on their content, gracefully displayed. Graphics do not become attractive and interesting through the addition of ornamental hatching and false perspective to a few bars. Chartjunk can turn bores into disasters, but it can never rescue a thin data set. The best designs (for example, Minard on Napoleon in Russia, Marey’s graphical train schedule, the cancer maps, the *Times* weather history of New York, the chronicle of the annual adventures of the Japanese beetle, the new view of the galaxies) are *intriguing and curiosity-provoking*, drawing the viewer into the wonder of the data, sometimes by narrative power, sometimes by immense detail, and sometimes by elegant presentation of simple but interesting data. But no information, no sense of discovery, no wonder, no substance is generated by chartjunk.

Forgo chartjunk, including
Moiré vibration,
the grid, and the duck.

One problem that has appeared repeatedly is that students have given us plots of n^2 and $n \lg n$ for n ranging from 1 to 10, to show the advantages of one of the $O(n \lg n)$ algorithms over bubble sort. There are two flaws: internal sorting of 10 elements is so fast that no one cares which algorithm is used, and for such short lists some of the $O(n^2)$ algorithms are faster than the $O(n \lg n)$ ones. If the plots showed from 100 to 10,000 records being sorted, the graph would be both more meaningful to the reader and more honest.

Also, the graphs are often misleading, in that they are labeled with seconds or some other inappropriate unit. If you had done some real speed tests, then seconds would be the right unit, but if you have just done a theoretical analysis, then it is better to be more explicit about what you are plotting—in most analyses the number would be the number of comparisons or the number of exchanges done. If you want to provide a good graph, you might look up the formula for the expected number of comparisons actually done by each algorithm. The best place to find accurate formulas for the standard sorting algorithms is Knuth’s *Sorting and Searching* [Knu73].

Here are some quick hints for creating usable graphs:

- Make sure that each graph has a single, clear purpose, and use the caption to point out what the reader is supposed to notice.
- Label both coordinate axes.
- Do not print a grid, but do put tick marks on the axes.
- Don’t clutter up the graph with cute pictures.
- Make sure the ranges of the axes are appropriate.

11.4.2 Pseudo-code

Most of you have seen flowcharts for describing techniques. Although touted as a cure-all to programming woes in the 1960’s, they have essentially disappeared from professional programming. They encouraged unstructured “spaghetti” code, and didn’t help much in understanding the resulting mess. The small boxes in flow charts give far too low an information density, using lots of paper to say very little.

Instead of flowcharts, most algorithms are now described using pseudo-code. The pseudo-code syntax can be based on any structured language (Algol, Pascal, C, Ada, . . .), but should be readable by someone familiar only with a different language. Tests and statements are not written in full detail, instead the intent of the statement or test is given. For example, pseudo-code for finding the minimum of an array might look like

```

min ← ∞
for each element of array
  if element < min then min ← element

```

Pseudo-code should be properly indented and either displayed like the example above, or put into a figure with a figure number and caption. Figures are generally easier for magazine and book typesetters to handle, and are essential for big chunks of pseudo-code, but displays are easier to read for small pieces of code.

If you do put the pseudo-code in a figure, try to arrange the layout so that the figure comes after the explanation has started. It is unfair to the reader to dump a big chunk of unexplained code in front of him or her. It can be very frustrating to slog through the code only to find an explanation after you turn the page.

11.5 Explaining recursion

Many students choose to describe a recursive algorithm, such as quicksort, for this assignment. Here are some hints on how to describe recursive algorithms so that they are comprehensible to someone who doesn't already know the algorithm, and who may not be entirely comfortable with recursion.

- First describe the outside view of the recursive function when viewed as a “black box” whose internal operations are not visible. For example, when describing a recursive procedure that sorts an array, say something like

The procedure `sort(arr, low, high)` sorts the elements of the array `arr[low]` through `arr[high]` into increasing order.

- Describe how the problem is subdivided and recombined in fairly general terms. For example,

Sorting is accomplished by dividing the sequence to be sorted into two smaller sequences, sorting each sequence independently, then merging the two sorted subsequences.

Note that the above description could apply equally well to quicksort or merge sort. The differences between the two algorithms come in how the partitioning and merging are done. Merge sort uses a trivial partition and a more complex merging operation, while quicksort uses a fairly complex partitioning operation and a trivial merge.

- Describe the boundary conditions that cause the recursion to stop.

A sequence with only a single element is already sorted, so no partitioning and merging are necessary. Fast variants of divide-and-conquer sorting algorithms often use insertion sorting when the sequence has fewer than five or six elements, as the overhead for recursion is quite high on conventional machines.

- Describe the details of the algorithm after giving the overview. Only after you've described the divide-and-conquer sorting strategy should you start giving the details that distinguish between merge sort and quicksort, and only after that should you start distinguishing between the different variants of quicksort. Here is the place to describe the importance of good pivot selection in quicksort, and to describe which of the many partitioning strategies you recommend.
- Do **not** attempt to walk through an execution of the code without explaining the structure. It is tempting to describe what happens in chronological order, but you'll lose the readers very quickly if they have to keep a six-deep recursion stack in their heads. Iterative algorithms are often best explained in chronological order, but recursive algorithms are not. If you are explaining quicksort, you may find it helpful to illustrate the partitioning step (which is iterative, not recursive) with pictures that show what operations happen, and in what order.

11.6 Titles, title pages, and executive summaries

When you prepare a formal report, it is worth taking a little extra care to make sure that the report will reach the intended audience, and that they will decide to read it. There are several things you can do to make it more likely that your report ends up in the right place:

- Use an informative title. If your report is titled *Quicksort* or *A report on certain problems within a database company*, no one will have any idea who to send the report to, where to file it, or whether it is worth reading. A title that a secretary or an executive can understand is much more valuable: *Speeding-up DDD's zip-code sorting by using quicksort*.
- Use a title page. Put the title and your name on a separate page from the body of the report. Not only does this look more professional than cramming everything into the body of the report, but it makes it easier for a busy executive to sort the reports he or she has to read. You can put your address and phone number on the title page also, so that anyone who wants to hire you again as a contractor can find you easily.
- Put an executive summary on a separate page from the body of the report. If the summary is short enough, it can go on the title page. The summary should tell the executives all they need to know about the report: what the problem is, what the solution is, and how much work will be needed to implement the solution, in just a few paragraphs. Remember that the executive may be handling dozens of reports for different problems in different products, and so a concise summary of the problem is as important as the solution.

11.7 Things to keep in mind for peer editing

11.7.1 About the algorithm

- This is a writing class, not an analysis of algorithms class. We don't really care which of the many decent sorting algorithms you choose. If you know something about sorting, there are a few obvious choices—quicksort, heapsort, and radix sort. Your explanation of the algorithm is far more important than which algorithm you choose. If you're clever, you'll choose an algorithm that is easy to explain, rather than the one that has the best asymptotic performance.
- When you read your partner's draft, check for bugs in the algorithm. Is the termination condition clearly and correctly specified? Does s/he explain how to be sure the program will not bomb when sorting zero entries? How are null strings handled? Could you take the description and write a procedure from it? Would the resulting procedure work?
- Particularly watch out for abuse of “big-O” notation. Many of you have just learned about asymptotic analysis, and tend to get carried away with its notations. The notation $f(n) = O(g(n))$ is badly designed—it is **not** an equality. Saying that $f(n)$ is $O(g(n))$ asserts a property of the function f , but does not give you a right-hand side that you can substitute for $f(n)$. The property is simply that, for sufficiently large n , $f(n)/g(n)$ remains bounded by some constant. Because the constants are all unspecified, it makes no sense to talk about $O(1.5n)$ or $O(n + 5)$ —both are just $O(n)$.

Frequently, students misinterpret a statement that an algorithm takes $O(n^2)$ steps to mean that it takes exactly n^2 . You can't claim that switching from an $O(n^2)$ algorithm to an $O(n \lg n)$ algorithm will give you a 100-fold speedup for $n > 1000$, because you don't know what the multiplicative constants are. The only way you can make such strong claims is to do a more detailed analysis of the algorithms.

If you look in Volume 3 of Knuth [Knu73, 107–110], you'll find that bubble-sort requires at most $\frac{1}{2}(n^2 - n)$ comparisons and $\frac{1}{2}(n^2 - n)$ exchanges to sort n items, and that on the average it takes $\frac{1}{4}(n^2 - n)$ exchanges and $\frac{1}{2}(n^2 - n \ln n - O(n))$ comparisons. All of these numbers are $O(n^2)$, but they do differ significantly. Two popular algorithms, heapsort and quicksort, are both $O(n \lg n)$ on the average, but heapsort is guaranteed to have fewer than $n \lceil \log_2 n \rceil$ comparisons [Knu73, 149], while quicksort could take as many comparisons as bubble sort. On random data, quicksort is usually the fastest sorting algorithm, but on non-random data it can be slower than bubble sort!

- You should also check the verbal translations of the big-O order notation. For example, “grows exponentially” has a specific technical meaning. The formula 2^n grows exponentially, but n^2 does not— n^2 grows quadratically.
- When you report the relative speeds of different algorithms, know what it is you are reporting! In most analyses of sorting routines, the number of comparisons or the number of exchanges is counted. Given the description of the problem, which is likely to be more important?
- When you cite further sources for an algorithm, make sure that the source you cite uses the same variant of the algorithm that you do. This is particularly important for algorithms like quicksort, where every author picks a slightly different variant. For quicksort, the main variations are in how the pivot element is chosen and in how the partitioning is done. The average behavior is $O(n \log n)$ for all the variants, but some of the variants are much easier to program or have less probable worst cases.

If you have trouble understanding one author’s version of quicksort, check another’s. But, please, don’t get two different versions mixed together! Warning: many authors pick a pivot for quicksort that gives the worst possible behavior with an already sorted list. Because zip code lists are likely to be re-sorted after a few new entries are added, you want to be sure that the algorithm you recommend will be fast even if the list is already sorted.

11.7.2 Mechanical details

- Check your partner’s use of articles. Particularly look for articles with names. For example, “quicksort” is the name of a sorting algorithm, so is used without an article. There is a substantial difference between implementing quicksort and implementing a quick sort.

If you are having trouble with definite and indefinite articles, read Chapters 29 and 30 of [HO91]. Particularly watch out for the word *data*; it used to be a plural noun (the plural of *datum*), but has come to be an uncountable one in American usage. Under no circumstances is it a singular, countable noun, so it can never be used with the indefinite article.

- Check your partner’s word choice for accuracy. For example, one popular phrase for this assignment is the oxymoron *slow speed*, which should be replaced with a phrase such as *slowness* or *low speed*. Another meaningless phrase that has been popular is *fast time*, which could have been replaced by *short time*, *little time*, *rapidly*, or *quickly*, depending on the context. You could also use more technical phrases such as *a shorter response time* or *better throughput*, but there is no point to being over-specific.
- Watch out for the back-formation *recurse* from *recursion*. The noun *recursion* comes from the verb *recur*. To recurse would be to swear at someone again. Verbs that add *-ation* to from nouns sometimes get corrupted by back-formation. The verbs that form *permutation* and *formation* are *permute* and *form*, not *permutate* and *formate*. Nouns with irregular plurals sometimes get corrupted also—the singular of *vertices* is *vertex*.
- Check your partner’s use of type styles carefully. For example, the names of variables and procedures are often done in a fixed-width typewriter font (such as Courier). If such a convention is used, it should be used consistently.
- Another common convention is to italicize important words when they are being defined. (Use underlining if you do not have italics on your typewriter or printer—do not use both underlining and italics.) For example,

A *divide-and-conquer* sorting algorithm has three steps: *partitioning* the sequence into two shorter sequences, recursively sorting each subsequence, and *merging* the two sorted subsequences to form a complete sorted sequence.

Italicizing words tells the reader that you believe the word is important, that the particular use you are making of the word may be unfamiliar to the reader, and that you are clarifying it here. Do not italicize jargon words every time you use them, but *only* when you are defining them. Italics are also used for foreign words that have not yet been Anglicized, and for general emphasis. In some documents, italics are also used for comments on program fragments, to make it clear that the comments are not part of the program itself.

- If you use the same typographic convention with many different meanings, it ceases to be helpful to the reader, so try to pick just a few related meanings for each typographic convention you use.

11.8 The final draft

Your report will be read by executives who understand the problem, but not programming. The format of the report can aid them considerably in finding the information they need. A jargon-free executive summary at the beginning, explaining the problem and its solution, may be all they need to read.

Use good section headings and captions for your figures. Read Sections 9.3 and 9.4 of Huckin and Olsen [HO91, 176–183] for suggestions about formatting a report for maximum readability.

Prepare your report as neatly as you can. A professional consulting job may produce a 10–20 page report from a \$2000 study. At more than \$100 a page, a consultant can afford to do it right. You do *not* need to spend enormous amounts on type-setting or professional illustrators. A typescript is still acceptable in industry, though laser-printing is becoming the standard, rather than the exception. Figures can be prepared on separate pages, pasted in, and photocopied. Photoreducing hand-drawn figures before pasting them in can hide the wiggly lines somewhat. The software available on the Macintosh and Windows machines makes combining graphics and text easy.

Choose an appropriate font size for your document. Unless you are writing a document that you don't want anyone to read (a warranty, perhaps?), choose at least a 10-point font. Unless you are writing for young children or people with bad eyesight, use at most a 12-point font. Standard elite and pica typewriters fall in the correct range. Using too large a font usually strikes readers as childish, which is not the effect you want in a report that you are charging hundreds of dollars for. If you want to impress someone with the bulk of a report, use 24-pound rag bond paper, a moderately wide font (say Palatino or Bookman instead of Times), slightly wider margins, and slightly more leading (space between lines), not 14-point fonts.

Check your spelling and punctuation carefully. Check even the spaces around your punctuation:

- Have two spaces after each colon, question mark, exclamation point, or sentence-ending period. (This is typewriter advice—when typesetting with variable-width fonts, no extra space is used after these punctuation marks.)
- Have spaces outside, but not inside parentheses and quote marks.
- Have no spaces before any punctuation marks except “(” and open-quote marks.
- Have no spaces around dashes.

Because of the difficulty students have had in the past describing sorting algorithms, we have requested two drafts with peer-editing on each draft before the final draft is due.