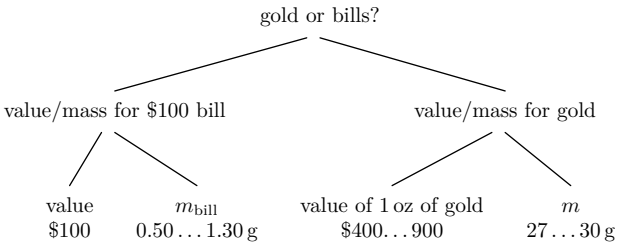
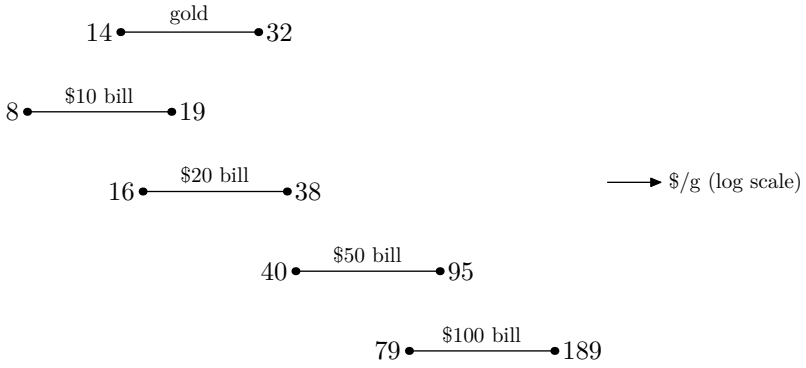


The next estimate is the value per mass of gold. I can be as accurate as I want in converting from ounces to grams. But I'll be lazy and try to remember the value while including uncertainty to reflect the fallibility of memory; let's say that $1\text{oz} = 27 \ldots 30\,\text{g}$. This range spans only a factor of 1.1, but the value of an ounce of gold will have a wider plausible range (except for those who often deal with financial markets). My range is \$400. . . 900. The mass and value ranges combine to give \$14. . . 32/g as the range for gold.

Here is a picture comparing the range for gold with the ranges for US currency denominations:



Looking at the locations of these ranges and overlaps among them, I am confident that the \$100 bills are worth more (per mass) than gold. I am reasonably confident that \$50 bills are worth more than gold, undecided about \$20 bills, and reasonably confident that \$10 bills are worth less than gold.

2.7 Example 4: The UNIX philosophy

The preceding examples illustrate how divide and conquer enables accurate estimates. An example remote from estimation – the design principles of the UNIX operating system – illustrates the generality of this tool.

UNIX and its close cousins such as GNU/Linux operate devices as small as cellular telephones and as large as supercomputers cooled by liquid nitrogen. They constitute the world's most portable operating system. Its success derives not from marketing – the most succesful variant, GNU/Linux,

is free software and owned by no corporation – but rather from outstanding design principles.

These principles are the subject of *The UNIX Philosophy* [14], a valuable book for anyone interested in how to design large systems. The author isolates nine tenets of the UNIX philosophy, of which four – those with comments in the following list – incorporate or enable divide-and-conquer reasoning:

1. Small is beautiful. In estimation problems, divide and conquer works by replacing quantities about which one knows little with quantities about which one knows more (**Section 2.5**). Similarly, hard computational problems – for example, building a searchable database of all emails or web pages – can often be solved by breaking them into small, well-understood tasks. Small programs, being easy to understand and use, therefore make good leaf nodes in a divide-and-conquer tree (**Section 2.3**).
2. Make each program do one thing well. A program doing one task – only spell-checking rather than all of word processing – is easier to understand, to debug, and to use. One-task programs therefore make good leaf nodes in a divide-and-conquer trees.
3. Build a prototype as soon as possible.
4. Choose portability over efficiency.
5. Store data in flat text files.
6. Use software leverage to your advantage.
7. Use shell scripts to increase leverage and portability.
8. Avoid captive user interfaces. Such interfaces are typical in programs for solving complex tasks, for example managing email or writing documents. These monolithic solutions, besides being large and hard to debug, hold the user captive in their pre-designed set of operations.

   In contrast, UNIX programmers typically solve complex tasks by dividing them into smaller tasks and conquering those tasks with simple programs. The user can adapt and remix these simple programs to solve problems unanticipated by the programmer.
9. Make every program a filter. A filter, in programming parlance, takes input data, processes it, and produces new data. A filter combines easily with another filter, with the output from one filter becoming the input for the next filter. Filters therefore make good leaves in a divide-and-conquer tree.

As examples of these principles, here are two UNIX programs, each a small filter doing one task well:

- `head`: prints the first lines of the input. For example, `head` invoked as `head -15` prints the first 15 lines.
- `tail`: prints the last lines of the input. For example, `tail` invoked as `tail -15` prints the last 15 lines.

▶ *How can you use these building blocks to print the 23rd line of a file?*

This problem subdivides into two parts: (1) print the first 23 lines, then (2) print the last line of those first 23 lines. The first subproblem is solved with the filter `head -23`. The second subproblem is solved with the filter `tail -1`.

The remaining problem is how to hand the second filter the output of the first filter – in other words how to combine the leaves of the tree. In estimation problems, we usually multiply the leaf values, so the combinator is usually the multiplication operator. In UNIX, the combinator is the **pipe**. Just as a plumber's pipe connects the output of one object, such as a sink, to the input of another object (often a larger pipe system), a UNIX pipe connects the output of one program to the input of another program.

The pipe syntax is the vertical bar. Therefore, the following **pipeline** prints the 23^{rd} line from its input:

```
head -23 | tail -1
```

But where does the system get the input? There are several ways to tell it where to look:

1. Use the pipeline unchanged. Then `head` reads its input from the keyboard. A UNIX convention – not a requirement, but a habit followed by most programs – is that, unless an input file is specified, programs read from the so-called standard input stream, usually the keyboard. The pipeline

   ```
   head -23 | tail -1
   ```

   therefore reads lines typed at the keyboard, prints the 23^{rd} line, and exits (even if the user is still typing).

2. Tell `head` to read its input from a file – for example from an English dictionary. On my GNU/Linux computer, the English dictionary is the file `/usr/share/dict/words`. It contains one word per line, so the following pipeline prints the 23^{rd} word from the dictionary:

   ```
   head -23 /usr/share/dict/words | tail -1
   ```

3. Let `head` read from its standard input, but connect the standard input to a file:

   ```
   head -23 < /usr/share/dict/words | tail -1
   ```

   The < operator tells the UNIX command interpreter to connect the file `/usr/share/dict/words` to the input of `head`. The system tricks `head` into thinking its reading from the keyboard, but the input comes from the file – without requiring any change in the program!

4. Use the `cat` program to achieve the same effect as the preceding method. The `cat` program copies its input file(s) to the output. This extended pipeline therefore has the same effect as the preceding method:

   ```
   cat /usr/share/dict/words | head -23 | tail -1
   ```

   This longer pipeline is slightly less efficient than using the redirection operator. The pipeline requires an extra program (`cat`) copying its input to its output, whereas the redirection operator lets the lower level of the UNIX system achieve the same effect (replumbing the input) without the gratuitous copy.

As practice, let's use the UNIX approach to divide and conquer a search problem:

▶ *Imagine a dictionary of English alphabetized from right to left instead of the usual left to right. In other words, the dictionary begins with words that end in 'a'. In that dictionary, what word immediately follows* `trivia`*?*

This whimsical problem is drawn from a scavenger hunt [29] created by the computer scientist Donald Knuth, whose many accomplishments include the TEX typesetting system used to produce this book.

The UNIX approach divides the problem into two parts:

1. Make a dictionary alphabetized from right to left.
2. Print the line following 'trivia'.

The first problem subdivides into three parts:

1. Reverse each line of a regular dictionary.
2. Alphabetize (sort) the reversed dictionary.
3. Reverse each line to undo the effect of **step 1**.

The second part is solved by the UNIX utility `sort`. For the first and third parts, perhaps a solution is provided by an item in UNIX toolbox. However, it would take a long time to thumb through the toolbox hoping to get lucky: My computer tells me that it has over 8000 system programs.

Fortunately, the UNIX utility `man` does the work for us. `man` with the `-k` option, with the 'k' standing for keyword, lists programs with a specified keyword in their name or one-line description. On my laptop, `man -k reverse` says:

```
  $ man -k reverse
  col (1)              - filter reverse line feeds from in-
put
  git-rev-list (1)     - Lists commit objects in reverse chrono-
logical order
  rev (1)              - reverse lines of a file or files
  tac (1)              - concatenate and print files in re-
verse
  xxd (1)              - make a hexdump or do the reverse.
```

Understanding the free-form English text in the one-line descriptions is not a strength of current computers, so I leaf through this list by hand – but it contains only five items rather than 8000. Looking at the list, I spot `rev` as a filter that reverses each line of its input.

▶ *How do you use `rev` and `sort` to alphabetize the dictionary from right to left?*

Therefore the following pipeline alphabetizes the dictionary from right to left:

```
rev < /usr/share/dict/words | sort | rev
```

The second problem – finding the line after 'trivia' – is a task for the pattern-searching utility `grep`. If you had not known about `grep`, you might find it by asking the system for help with `man -k pattern`. Among the short list is

```
grep (1)             - print lines matching a pattern
```

In its simplest usage, `grep` prints every input line that matches a specified pattern. For example,

```
grep 'trivia' < /usr/share/dict/words
```

prints all lines that contain `trivia`. Besides `trivia` itself, the output includes `trivial`, `nontrivial`, `trivializes`, and similar words. To require that the word match `trivia` with no characters before or after it, give grep this pattern:

```
grep '^trivia$' < /usr/share/dict/words
```

The patterns are **regular expressions**. Their syntax can become arcane but their important features are simple. The ^character matches the beginning of the line, and the $ character matches the end of the line. So the pattern ^trivia$ selects only lines that contain exactly the text `trivia`.

▶ *This invocation of `grep`, with the special characters anchoring the beginning and ending of the lines, simply prints the word that I specified. How could such an invocation be useful?*

That invocation of `grep` tells us only that `trivia` is in the dictionary. So it is useful for checking spelling – the solution to a problem, but not to our problem of finding the word that follows `trivia`. However, Invoked with the `-A` option, `grep` prints lines following each matching line. For example,

```
grep -A 3 '^trivia$' < /usr/share/dict/words
```

will print 'trivia' and the three lines (words) that follow it.

```
trivia
trivial
trivialities
triviality
```

To print only the word after 'trivia' but not 'trivia' itself, use `tail`:

```
grep -A 1 '^trivia$' < /usr/share/dict/words | tail -1
```

These small solutions combine to solve the scavenger-hunt problem:

```
rev </usr/share/dict/words | sort | rev | grep -A 1 '^trivia$' | tail -1
```

▶ *Try it on a local UNIX or GNU/Linux system. How well does it work?*

Alas, on my system, the pipeline fails with the error

```
rev: stdin: Invalid or incomplete multibyte or wide character
```

The `rev` program is complaining that it does not understand a character in the dictionary. `rev` is from the old, ASCII-only days of UNIX, when each character was limited to one byte; the dictionary, however, is a modern one and includes Unicode characters to represent the accented letters prevalent in European languages.

To solve this unexpected problem, I clean the dictionary before passing it to `rev`. The cleaning program is again the filter `grep` told to allow through only pure ASCII lines. The following command filters the dictionary to contain words made only of unaccented, lowercase letters.

```
grep '^[a-z]*$' < /usr/share/dict/words
```

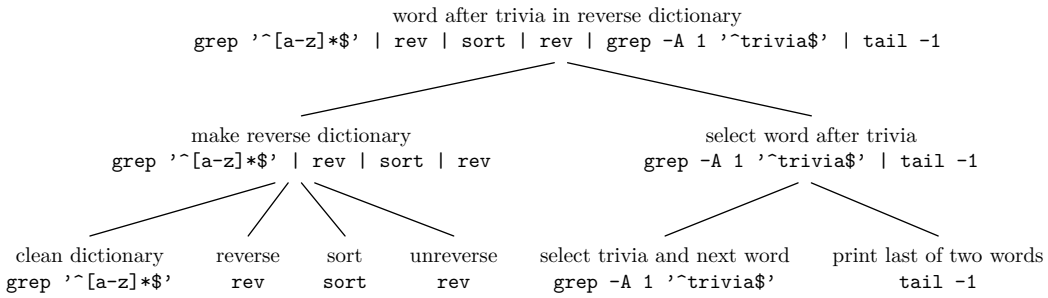
This pattern uses the most important features of the regular-expression language. The ^ and $ characters have been explained in the preceding examples. The `[a-z]` notation means 'match any character in the range a to z – i.e. match any lowercase letter.' The `*` character means 'match zero or more occurrences of the preceding regular expression'. So ^[a-z]*$ matches any line that contains only lowercase letters – no Unicode characters allowed.

The full pipeline is

```
grep '^[a-z]*$' < /usr/share/dict/words \
| rev | sort | rev \
| grep -A 1 '^trivia$' | tail -1
```

where the backslashes at the end of the lines tell the shell to continue reading the command beyond the end of that line.

The tree representing this solution is



Running the pipeline produces produces 'alluvia'.

Problem 2.11 Angry
In the reverse-alphabetized dictionary, what word follows angry?

Although solving this problem won't save the world, it illustrates how divide-and-conquer reasoning is built into the design of UNIX. In short, divide and conquer is a ubiquitous tool useful for estimating difficult quantities or for designing large, successful systems.

Main messages

This chapter has tried to illustrate these messages:

1. Divide large, difficult problems into smaller, easier ones.
2. Accuracy comes from subdividing until you reach problems about which you know more or can easily solve.
3. Trees compactly represent divide-and-conquer reasoning.
4. Divide-and-conquer reasoning is a cross-domain tool, useful in text processing, engineering estimates, and even economics.

By breaking hard problems into comprehensible units, the divide-and-conquer tool helps us organize complexity. The next chapter examines its cousin abstraction, another way to organize complexity.

Problem 2.12 Air mass
Estimate the mass of air in the 6.055J/2.038J classroom and explain your estimate with a tree. If you have not seen the classroom yet, then make more effort to come to lecture (!); meanwhile pictures of the classroom are linked from the course website.

Problem 2.13 747
Estimate the mass of a full 747 jumbo jet, explaining your estimate using a tree. Then compare with data online. We'll use this value later this semester for estimating the energy costs of flying.

Problem 2.14 Random walks and accuracy of divide and conquer
Use a coin, a random-number function (in whatever programming language you like), or a table of reasonably random numbers to do the following experiments or their equivalent.

The first experiment:

1. Flip a coin 25 times. For each heads move right one step; for each tails, move left one step. At the end of the 25 steps, record your position as a number between -25 and 25.
2. Repeat the above procedure four times (i.e. three more times), and mark your four ending positions on a number line.

The second experiment:

1. Flip a coin once. For heads, move right 25 steps; for tails, move left 25 steps.
2. Repeat the above procedure four times (i.e. three more times), and mark your four ending positions on a second number line.

Compare the marks on the two number lines, and explain the relation between this data and the model from lecture for why divide and conquer often reduces errors.

Problem 2.15 Fish tank

Estimate the mass of a typical home fish tank (filled with water and fish): a useful exercise before you help a friend move who has a fish tank.

Problem 2.16 Bandwidth

Estimate the bandwidth (bits/s) of a 747 crossing the Atlantic filled with CDROM's.

Problem 2.17 Repainting MIT

Estimate the cost to repaint all indoor walls in the main MIT classroom buildings. [with thanks to D. Zurovcik]

Problem 2.18 Explain a UNIX pipeline

What does this pipeline do?

```
ls -t | head | tac
```

[Hint: If you are not familiar with UNIX commands, use the `man` command on Athena or on any nearby UNIX or GNU/Linux system.]

Problem 2.19 Design a UNIX pipeline

Make a pipeline that prints the ten most common words in the input stream, along with how many times each word occurs. They should be printed in order from the the most frequent to the less frequent words. [Hint: First **t**ranslate any non-alphabetic character into a newline. Useful utilities include `tr` and `uniq`.]
