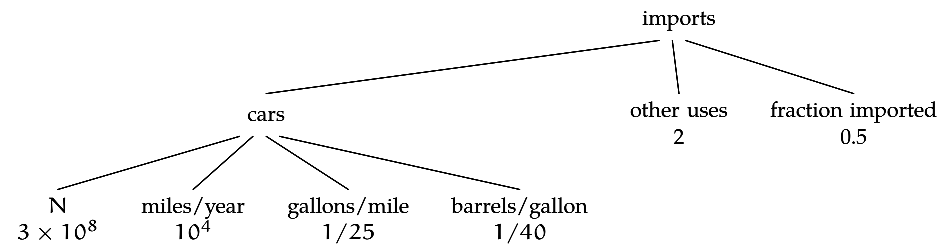


All the leaves have values, so I can propagate upward to the root. The main operation is multiplication. For the 'cars' node:

$$3 \times 10^8 \text{ cars} \times \frac{10^4 \text{ miles}}{1 \text{ car–year}} \times \frac{1 \text{ gallon}}{25 \text{ miles}} \times \frac{1 \text{ barrel}}{40 \text{ gallons}} \sim 3 \times 10^9 \text{ barrels/year}.$$

The two adjustment leaves contribute a factor of $2 \times 0.5 = 1$, so the import estimate is

$$3 \times 10^9 \text{ barrels/year}.$$

For 2006, the true value (from the US Dept of Energy) is 3.7×10^9 barrels/year – only 25 higher than the estimate!

Problem 1.5 Midpoints
The midpoint on the log scale is also known as the geometric mean. Show that it is never greater than the midpoint on the usual scale (which is also known as the arithmetic mean). Can the two midpoints ever be equal?

1.5 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 successful 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* [13], a valuable book for anyone interested in how to design large systems. The author

Global comments

I'm having a hard time reading this section (I have no background really in coding, nor do I use UNIX). I feel like I'm going to have to read this a bunch of times before I begin to understand what is actually going on. I understand the use of divide in conquer in this context, I'm just mostly confused following these examples (considering how easy and great the previous sections were to read).

I agree with this. There is so much technical jargon in this section that most of my time is spent trying to understand and remember the new terms/commands instead of divide and conquer.

Agreed - I suggested there be a short description of how UNIX works preceding this chapter. I don't have much UNIX experience either and was quite confused, so I imagine everyone else in the same boat would have similar problems.

I feel the exact same way....divide and conquer permeates all of this section, but that's the extent of my understanding.

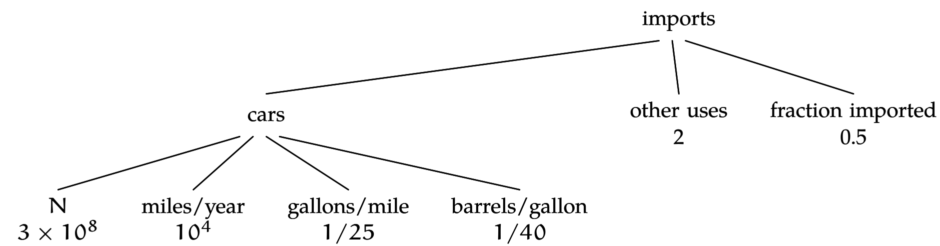
I agree with previous comments; it would also be helpful if you defined or explained some of the terms being used in this section – utility, input/output, like you did for "filter"

was UNIX the first program with this kind of search function (ie. searching for trivial, nontrivial, trivializes along with trivial)? This seems to be pretty ubiquitous with all searches nowadays.

Wikipedia says that there was earlier precedents. But UNIX seems to be the first system to use regular expressions everywhere. A nice feature of the UNIX approach is that the grep program gives you back lines, on which you can do more things (like in class when we extracted the email addresses from the Stellar HTML file).

This section requires a foundational knowledge of UNIX that I don't think most people have...the overall message of divide and conquer is being lost amongst all these functions that are being used.

I agree to a certain extent. While I'm familiar with Unix, I could imagine this section discouraging those who do not. Is it possible to illustrate the concepts another way?



All the leaves have values, so I can propagate upward to the root. The main operation is multiplication. For the 'cars' node:

$$3 \times 10^8 \text{ cars} \times \frac{10^4 \text{ miles}}{1 \text{ car–year}} \times \frac{1 \text{ gallon}}{25 \text{ miles}} \times \frac{1 \text{ barrel}}{40 \text{ gallons}} \sim 3 \times 10^9 \text{ barrels/year}.$$

The two adjustment leaves contribute a factor of $2 \times 0.5 = 1$, so the import estimate is

3×10^9 barrels/year.

For 2006, the true value (from the US Dept of Energy) is 3.7×10^9 barrels/year – only 25 higher than the estimate!

Problem 1.5 Midpoints
The midpoint on the log scale is also known as the geometric mean. Show that it is never greater than the midpoint on the usual scale (which is also known as the arithmetic mean). Can the two midpoints ever be equal?

1.5 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 successful 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* [13], a valuable book for anyone interested in how to design large systems. The author

I think the UNIX example not broad enough for someone unfamiliar with the UNIX system to understand quickly. I like examples that are much more general/things everyone encounters on a fairly regular basis much better.

I agree with this comment completely. I could follow the example because I have some experience with UNIX. However, I feel like this would have been fairly hard to follow for someone with no UNIX experience, especially since a large part of the people in this class are not course 6.

I think the unix example is a good example of how divide and conquer can be used for more complex problems and how it is applicable outside the subject matter. However, I think that the heavy programming language necessary to make this point distracts readers from the main messages. This chapter seems more like an in depth example than a lesson. Is that what you were going for?

Is that why the tree diagrams focus on one thing in the larger spectrum of example? For example, when calculating the imported oil consumption in the U.S., the estimation was also based on the barrals of oil used for cars.

I have no idea what this means....hold the user captive in their pre-designed set of operations? How is that related to email or writing a document?

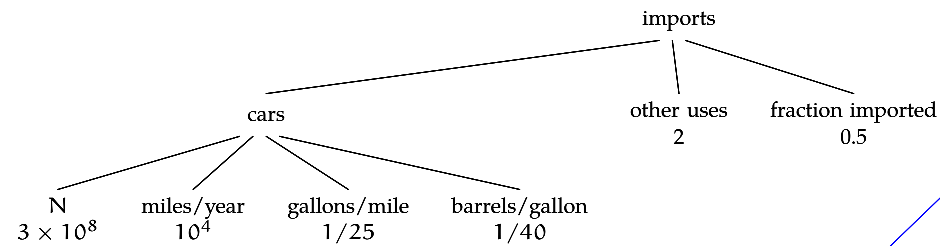
Ok. I think this has finally made sense. I took Linguists and it definitely makes it easier to model and understand this tree diagrams. As well as the ones we make via actual calculations using divide and conquer

Unlike the other divide and conquer problems seen in this class, this method applied to Unix in this specific example does not seem very useful. The question of having a backwards dictionary seems improbable in real life.

Will we be using other methods to solve this Unix problem over the course of the class?

I don't really like this as a largescale example, I feel like I need to know more about UNIX to understand it better.

I had a difficult time understanding this article. I feel that the last example was much more straightforward. This seems like it requires a decent level of knowledge of UNIX to understand anything more than the very basic idea that large problems need to be broken into smaller problems.



All the leaves have values, so I can propagate upward to the root. The main operation is multiplication. For the 'cars' node:

$$3 \times 10^8 \text{ cars} \times \frac{10^4 \text{ miles}}{1 \text{ car–year}} \times \frac{1 \text{ gallon}}{25 \text{ miles}} \times \frac{1 \text{ barrel}}{40 \text{ gallons}} \sim 3 \times 10^9 \text{ barrels/year}.$$

The two adjustment leaves contribute a factor of $2 \times 0.5 = 1$, so the import estimate is

$$3 \times 10^9 \text{ barrels/year}.$$

For 2006, the true value (from the US Dept of Energy) is 3.7×10^9 barrels/year – only 25 higher than the estimate!

Problem 1.5 Midpoints
The midpoint on the log scale is also known as the geometric mean. Show that it is never greater than the midpoint on the usual scale (which is also known as the arithmetic mean). Can the two midpoints ever be equal?

1.5 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 successful 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* [13], a valuable book for anyone interested in how to design large systems. The author

Comments on page 1

Read Section 1.5 for Tuesday's lecture (due to the Monday holiday). The memo is due by Monday at 10pm. Don't worry about the last two pages – they are a bunch of end-of-chapter problems (but some of them appear on your first homework).

This example really hit home what trees and divide and conquering were, how to use them, and the thinking process of breaking down big problems into manageable pieces. Because this example did not include "speaking to your gut" it was much more digestable and I would have liked to have seen this example proceed the estimation divide and conquering problems.

I also feel this way. I supposed partly because this is a course 6 class, and we are mostly engineers, that tis kind of example would really hit home with us.

I like your use of "speaking to your gut" and "digestible"

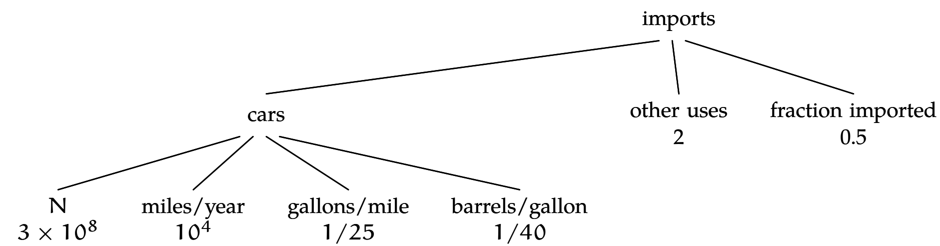
What makes one operating system "a cousin" of another? Are they simply operating systems that were created by the same people? Or those with similar features? Is there any formal classification?

Does an example not actually about estimation really have place in a book about estimation? I believe you made your point about how useful divide and conquer is when you first introduced the idea.

Is it truly portable? (I don't know anything about computers.) For example, with regards to web browsers, I've heard that either Chrome of Firefox is portable, but some of its features take data stored in IE, since it is assumed that all computers have IE. Are there conditions/pre-existing features necessary for UNIX/GNU/Linux to run?

It's important to keep in mind that "most successful" is still not particularly successful in comparison to corporate owned operating systems, due to both marketing and ease of use.

The point here is just that Linux is the most successful variant of UNIX, and it's not due to marketing. Your comment depends on your definition of success. I wouldn't want most research servers running Windows, even if it's a useful OS for home and business users.



All the leaves have values, so I can propagate upward to the root. The main operation is multiplication. For the 'cars' node:

$$3 \times 10^8 \text{ cars} \times \frac{10^4 \text{ miles}}{1 \text{ car–year}} \times \frac{1 \text{ gallon}}{25 \text{ miles}} \times \frac{1 \text{ barrel}}{40 \text{ gallons}} \sim 3 \times 10^9 \text{ barrels/year}.$$

The two adjustment leaves contribute a factor of $2 \times 0.5 = 1$, so the import estimate is

$$3 \times 10^9 \text{ barrels/year}.$$

For 2006, the true value (from the US Dept of Energy) is 3.7×10^9 barrels/year – only 25 higher than the estimate!

Problem 1.5 Midpoints
The midpoint on the log scale is also known as the geometric mean. Show that it is never greater than the midpoint on the usual scale (which is also known as the arithmetic mean). Can the two midpoints ever be equal?

1.5 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 successful 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* [13], a valuable book for anyone interested in how to design large systems. The author

Is this also true of UNIX, or is UNIX actually owned by a corporation?

As I understand it, Unix, while an owned trademark, refers mainly to a standardization. Whereas Linux is an actual open-source OS.

I think UNIX was developed partially at Bell Labs

I believe Unix IP has ownership; that is in fact the impetus behind Linux development. An apocryphal(?) story is that it was named because it was a "Unix Like" kernel.

I believe the Apple OS is also UNIX-based (though Windows is not)

I'm also a little unsure of how UNIX works and could use a little more info about how it is related to GNU/Linux as cousins.

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 7.3). 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 1.3).
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.

In a discussion of estimation, why include the philosophy points that don't pertain to divide & conquer? You could address the 4 related points first and then mention the others, but having some with comments and some without seemed a bit jarring to me.

I also found them somewhat distracting, especially because without some explanation, I really have no idea what they mean. Each of them raises questions instead of furthering my understanding of divide and conquer.

I agree, the inclusion of the other 5 points is unnecessary and confusing. I found myself constantly evaluating whether I was currently reading one of the 4 important points. Also, to those not from a computer science background, these other points would only confuse them, even though they're not important to the later discussion.

I think the other 5 tenets were included for completion (why leave out parts of a set?), but I agree that for clarity they should be dropped.

Is this itself an example of dividing and conquering?

I would say no in the estimation since, as there is no number problem being solved with these techniques; however, as has been stressed many times, these skills aren't just useful for the purposes of guestimating numbers. It can be applied to a variety of approaches to thinking, in which case, yes, it is?

No, it's identifying general themes about UNIX, so it's more about lumping common aspects together. The 9 concepts aren't separate parts that you could put together to make UNIX.

Philosophy related to divide and conquer? That seems a bit of an odd parallel. Maybe structure makes more sense than philosophy?

I don't think this is limited to d&c. It's not a structure, it's a set of design principles, which can be referred to as a (design) philosophy.

What do you mean by comments? It seems like all of them have comments.

I think 3 through 7 are just the tenets, without comments. I agree that it's not particularly clear what constitutes the added comment. I think though, that just the first sentence of each item is the tenet itself.

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 7.3). 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 1.3).
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.

The lack of memory in early UNIX systems also played a large factor in the division of programs into smaller pieces as each could only access something on the level of 64K of memory - not enough to do complex operations as a single program but requiring multiple programs with very specialized function. From other UNIX history readings I've had for other classes it seems less of a "UNIX made for divide and conquer" and more of a "divide and conquer happens to work well with UNIX"

I would like to see the actual principles set apart from the commentary somehow. Perhaps making the principles bold would do the trick?

agreed.

And/or having a line break after the principle so it's on its own line before the comments.

I think this is a very useful principle in divide and conquer that should have been stated at the very beginning. It helps me to think about what sort of categories to divide into to make the estimation easier for me.

I understand that this is the way to go. The hard part is realizing you know something about things you don't have direct knowledge about. I found this the most frustrating part of the diagnostic.

I agree, this could be the first sentence in the divide and conquer section. I have talked to people in the class and thought that you knew a lot of facts they wouldn't when the important part is that you need to isolate things you know.

Should this actually read Section 1.3?

Yeah, typo most likely.

This is information that could have been stated earlier when describing "divide-and-conquer"

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 7.3). 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 1.3).
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.

Would this also improve efficiency, because now the computer can search in parallel instead of in serial for each leave? Or no?

Hmm I'm curious about this as well - would it make things faster too?

Computers can only search in parallel, if they have more than one designated processor. These days, you hear about "dual-core" or "quad-core" processors. Those can do parallel computing. Otherwise, smaller programs won't make computing faster. They'll just make it easier for you to debug and write the overall program.

In fact, having more, smaller programs could slow things down as more things would have to be loaded into memory, taking longer than one longer single load.

This is similar to constraint based design in Mechanical engineering. The idea is you choose the degrees of freedom you don't want and then only place constraints that kill the individual degrees of freedom. This way you're not overconstrained which often leads to mechanical failure.

This is the idea of modularity, right? If we keep things simple and contained it will be easier to work with.

I think it would be nice to state modularity in the description

Modularity doesn't just work on this simplest level, but also at higher levels where parts are connected. Connections and independence between parts are also key to modularity. You can have unitaskers that aren't interchangeable. I agree that it might be a theme, but it's not this one exactly.

use of modularity in programming

is this saying that we should make clear trees? I am not getting the connection to divide and conquer

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 7.3). 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 1.3).
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.

I'm not sure I understand what the difference between 1. and 2. is? Both seem to break up a large task into small tasks.

I think one is about partitioning where things go and how you find them (1) while the other is about actually running scripts to do tasks, like work processing. At least, thats what I understood.

The way I saw it was: 1. Make the programs small. 2. Make the programs simple. I guess they could be combined, but for some reason my intuition tells me that it's easier to understand as two separate points.

Yeah, I agree. The way I see it, in the first point the author is saying break complex tasks into smaller ones, so each individual task is easier to understand and solve. And then in the second point he's saying specifically to make each program do only one task, and that this way, the program is easier to debug

In addition to being simpler and easier to debug, the main point is that since its one task, you can clear you mind and make that one task done well. A small program is fine, but if it still isn't done well then its pointless. After making a bunch of small program that work very effectively, chances are when you put them together they will work effectively as well.

Another thing that is nice about small, simpler specialized programs is that they can be used in many contexts. With the example of spell-checking, it can be used by more than one application (Word, email etc)

How does this apply to the need for more processes to be run concurrently?

Do you mean more processes need to be run to accomplish a complex task? That could be a problem, as stated in an earlier comment, when it comes to processing power, but the small processes wouldn't be running concurrently, they'd be running in series, which may cause a slowdown.

I didn't understand which of the nine points were the four incorporating divide and conquer until I read the comments. Even then, I had to assume that the four with explanations were divide and conquer examples

Go into more detail- how is this using divide and conquer?

I don't know if this is supposed to be one of the four, but I think this is roughly applicable if you substitute "prototype" with "constructing problem solving formulas"

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 7.3). 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 1.3).
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.

Prototypes are usually for testing a particular piece of your idea, independent of the rest of the structure, which is the fundamental idea behind divide and conquer. This point should be elaborated a bit more.

I don't really understand how this is part of a philosophy. isn't it the equivalent of don't procrastinate?

No, I think that what they mean is that it's more important to have a prototype out that's 70% done in 2 weeks than an almost finalized product that's 95% done in 2 months because user response is valuable in redirecting projects and allows for a more intimate iterative process. It's less about procrastination and more about understanding that it's important to get a prototype out there asap for the users to say what they like and dislike.

I took this to be a philosophy for the developer to get off the ground, rather than getting something out there to be tested by users. I interpreted it as how it's much easier for me to edit something I wrote poorly than write something new. Therefore, I should just write something rather than stare at the blank paper.

in upop, we learned something very similar to this concept, and called it, "make mistakes faster"

Hopefully an iterative spiral model, with prototypes being built starting from lower fidelity prototypes to higher fidelity.

This is also applicable to rough estimates: ease over precision

Sort of, but aren't we all about efficiency (rounding to 1, few, and 10) AND portability (perhaps using pi*10^7 to remember seconds/yr)?

yes but if we had to choose between the two, we would choose portability; because that is the whole premise of estimation.

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 7.3). 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 1.3).
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.

I'm not sure I understand this, in the context or UNIX or in the context of divide and conquer.

I also am not so sure what they mean about choosing portability over efficiency. Could someone please explain this?

does portability mean less code rather than more efficient code?

I'm not really a coder, but my guess was that it's saying that it's better to have very clear, defined code, even if that means adding a couple extra lines.

yeah the use of portability is confusing.

I think this also goes along with being able to use the same code with multiple programs (back to spell checking, using it for word processing, email etc) instead of having each program have their own spell-checker (or whatever it is they might share)

I thought this concept of being able to use the same code with multiple programs was called modularity? But I also don't have a strong computer science background and might be wrong. But the post regarding extra lines for clear definition makes sense too.

I think portability is being mentioned for its benefit of being less specific to certain platforms, etc. Choosing simpler functions or methods more universal is more accepted than exploiting the most efficient approach on a given platform.

I also think that portability may be referring to modularity. Being able to you a small code for multiple things. Although, im not sure how they justify it being better than efficiency.

A tool that you can use for many different problems, that will get you close to the answer is much better then one that will give you one exact answer to only one problem.

I think this refers to the speed which is required in solving approximation problems. You want to come up with a plan of attack as soon as possible.

Who wrote these philosophies? Did a bunch of programmers came together and write down what they thought was important at the time?

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 7.3). 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 1.3).
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.

What is a "flat" text file? Does this imply using .rtf instead of something more complicated, or storing things just once instead of layering them over older versions of the same file?

I think it means plain ASCII files. Even RFT files have some formatting in them, and so require programs with special abilities to read. But any computer ever built can open and display an ASCII file readably without any special software.

I agree with the second post - I think a "flat" text file is basically one that doesn't require specialized software to open (RTF, etc).

i wouldn't have known this either, but it seems to not matter in this context of divide-and-conquer.

Just for the sake of discussion, I think "flat" can also be used to distinguish a file that can be read and understood sequentially (often ASCII) from a database system that has a non-standard underlying file/storage structure.

Even though these aren't relevant to divide and conquer I still might like to see a sentence describing what is meant by each.

I feel that the reasons it is not very easy to understand this it's because very little has to do with Divide and Conquer. reading the next pages helps to understand it better

What is Flat?

more details about these would be helpful

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 7.3). 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 1.3).
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.

What is software leverage?

I also have no idea. Maybe simply that there is a lot of free software out there that one can incorporate into UNIX?

I was unclear about the meaning also. After looking it up, I found a pretty good explanation of it here: http://books.google.com/books?id=MFzVYQsD60AC&lpg=PA67&o xiEl9bI&dq=%22software%20leverage%22%20definition&pg=PA67#v=onepage&c . It is a bit of a read and would have been better had the term been explained here.

I didn't understand a lot of the phrases in 1-9. Perhaps an easier concept on something less technical, or more definitions. I understand the divide & conquer parts, but thinking about the new words made it difficult to not get overwhelmed.

I agree, I understand the divide and conquer parts, but I had to look up a lot of these terms. Maybe another sentence in 1-9 that explains these terms of gives an example would be helpful

Or just omitting 3 through 9? They don't seem to contribute much to the topic at hand, and many of us seem to be confused by them.

I'm not familiar with this terminology. What is "software leverage"? And how do you use it?

Agreed - this is confusing for me as well.

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 7.3). 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 1.3).
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.

I'm not sure what a shell script is - since it seems that there's some confusion over some UNIX specific points, perhaps it would be best to open this section with a short description of how UNIX works?

Agreed; this section so far isn't so accessible if you don't know much about operating systems or programming, so maybe a bit more explanation would be nice (although maybe then it'd defeat the purpose of using this as a parallel to explain divide and conquer and just turn into a lesson on UNIX which isn't what you want either...)

Agreed. I looked up shell scripts on wikipedia: A shell script is a script written for the shell, or command line interpreter, of an operating system. It is often considered a simple domain-specific programming language. Typical operations performed by shell scripts include file manipulation, program execution, and printing text."

Shell scripts may be useful because they are typically portable within UNIX based operating systems. Without incorporating dependencies to specific software (ex. using flat txt files), the code is more likely to work on other UNIX based operating systems.

I thought a captive UI was sort of UI that disallows interactivity once the user starts the application...but I'm not sure how email is like that...

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 7.3). 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 1.3).
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.

What is a "captive user interface"? (it just doesn't sound pleasant...) Because of that, I don't really understand what is going on in this tenet.

Once again...agreed, I have no idea what this means either.

> p.s. If you use the "I agree" option, then this question (which clearly many people have) will get moderated upward and will be one of the first that I see when I look at all responses.
>
> Whereas NB is not (yet?) smart enough to do the same if you verbally agree. AI has a ways to go. In short, use the "I agree" option (or "I don't agree") if it captures what you want to say.

I'm not entirely sure but based on the rest of the paragraph I would guess they are limited GUI interfaces that don't allow maximum user freedom. Programs like MATLAB and photoshop (ignoring common filters) are what I would consider good examples of a non captive user interface, because they allow the user to define and modify operations to their liking.

I think "captive user interfaces" refer to super fancy layouts but simply for the glittery oo ahhh effect. Actually come to think about it, its probably referring to restrictive UIs, like the old DOS programs.

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 7.3). 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 1.3).
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.

You're right, it's not pleasant. Though it's common. For example, the typical word processing program allows only one way to edit and create your document: You must use its user interface or else. For example, how else can you make an OpenOffice or MS-Word document, except by using the corresponding program?

Whereas with a non-captive-UI, you could use any method you like to make a word-processing file, which is then (say) turned into PDF by the word processor. TeX is an example of this kind of program. I have many programs that themselves write TeX code, which is then compiled to make PDF.

The trees in this chapter are made by a similar method. The figure-drawing program is called MetaPost, which is much easier to program than directly writing PostScript. But even so, making the divide-and-conquer trees become a bit tedious because I kept repeating so much of the MetaPost code. So I wrote a that program takes a simple text-file representation of a tree and converts (compiles) it into MetaPost, which is then compiled into PostScript and PDF. That would be very hard to do using a graphical figure-drawing program. Because of the captive user interface, only a human typing and clicking at the keyboard and mouse can generate the drawing.

this clearly explains what captive user interface is. thanks for the examples. I think the ones you listed here are a little better than the ones in the reading.

Are any examples of this? I can understand that user interfaces are large and hard to debug, but how exactly would you divide and conquer this?

Again, some unfamiliar terminology.

I don't know what captive user interfaces are.

It seems like this principle has been violated by quite a few programs nowadays...

I would be interested to see an example of such a program because I'm still not really sure that i understand what is meant here.

...yeah but shitty interface repels the average, non-programmer user

i don't really understand how these two concepts are related, or how the unix really solves the problems of the first paragraph.

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 7.3). 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 1.3).
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.

I'm not sure if I'm understanding this correctly, but doesn't leaving the program from the user to define tend to deter the user from using the program since it means extra work?

Not if you do it right, I think its good to give the user some control over the processes. As long as the smaller tasks are atomic the user should be able to do what they want without messing things up.

So they don't use UI? Or they have separate UIs for each program that can be combined into one?

I think it might be more that they are not limited to the buttons seen in like, typical email. They can either make their own or use others that the original designers didn't think of.

I agree... its not necessarily that a UI doesn't exist, but rather that if a UI module does exist, it doesn't limit functionality. Simple programs exist first, and a UI may be built on top of it to provide visual support, but may or may not incorporate all of the program's functionality / ability.

I'm not sure I completely follow this. I'm not familiar with the terms, but it seems like the author is being a bit repetitive. Points 1 and 2 were pretty similar–saying to split programs into smaller programs. Here, he's also saying the same thing–"solve complex tasks by dividing them into smaller tasks..."

This sounds to me more like robustness (not necessarily redundancy), and less like our goal of simplification when we use divide and conquer, right?

this is a much easier term to understand than the previous ones presented.

Works like a transfer function in systems and signals. If you can convert the elements into transfer functions, it's easy to design a system to match the desired performance requirements.

This is a superb definition of a program

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 7.3). 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 1.3).
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.

I like how you defined filter here, but I feel like filter is actually a more-known term than the ones you did not define above.

I agree. I like the definition, but some others are more necessary.

I agree too. I needed definitions for almost every term on this list.

I agree, too. This filter example makes sense to me. Basically, make your programs all mesh with each other–input for one is the output for another, so you don't have to keep using user input all over

I don't quite understand how this definition of filter makes it a good leaf in a tree...

Don't take "leaf" so literally; you may be thinking of leaves as the end of the branches, but I think most often leaves just mean another branching point. If you think of it that way, filters take data, process it, and then out put data for input into the next branch just like all the other branches we have talked about in other trees in class.

No – a leaf necessarily (by definition) has no children. I think the right term here would be "internal node", not leaf.

I agree – this doesn't make much sense the way it is written. The idea we seem to be going for is that, for example, multiplication and dimensional analysis combine easily in arbitrary ways, just like filters, but I'm not sure the way this is explained works.

Here you explain how filters relate to the idea of divide-and-conquer... but it's not so clear for some of the earlier points. I'm not sure if you're trying to just list the main tenets of the UNIX philosophy, or if each of these points is supposed to correlate to some aspect of divide and conquer.

It says in the line proceeding the list that only the 4 with explanations are relevant. Maybe including the others is unnecessary since they don't seem to have any real relevance to divide an conquer.

Agreed. Maybe some explanation of UNIX followed by just the tenets that relate to d&c would be clearer.

I think this is a very good informative section; I personally really enjoy sections like this because it's the little facts that make one a more worldly, learned individual. That said, it does seem to be a lot of info that isn't immediately relevant to estimation? On the other hand, I prefer it this way and I'm sure most MIT (and college) students do to, and that's the target audience.

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 7.3). 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 1.3).
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.

how is a program different from a filter otherwise? don't filters remove noise? but that's not the definition used here.

A filter seems to be a type of program, that may or may not remove noise. A filter, as it is used here, seems to be more of an operation.

I think you are thinking of a filter too literally, not just signal processing, but for example, a text editor would have a small program(more likely a function) that takes input from the keyboard and displays it on the screen. It filters the keystrokes into ASCII characters/pixels on your screen. I think that's what's meant by filter...

I agree. It would be very useful if you identified the four principals that apply to both programming and divide and conquer. Also, those four points should be parallel to one another in their explanations.

This section is hard to understand for someone with little to no programming knowledge. I'd like if more terms were defined or if programming was compared to divide and conquer as opposed to the opposite (which is how it seems now)

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 23rd 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
   ```

Comments on page 3

I'm a bit confused here, and in the following section: are we writing this somewhere, like code, or is this an already made program like, say, notepad?

head and tail are already programs that exist in UNIX, like in Athena the ls and cd commands that you are probably are familiar with. When they are combined below that is written by the user.

Would run be a simpler term?

This example for filters is helpful, but I feel like the author's point (#9) about filters was already the clearest. An example about #8 (captive user interfaces) would have been helpful too

I disagree I think the verbal explanation of a filter was pretty clear but this makes sure I understand it 100% and I'm glad it's here. I do agree that examples of the other ones would be very helpful

This looks a little strange. Maybe put in quotations the actual input, or make it a different font, or something.

How you arranged it below, by putting it on a separate line, would work well too

I would either put it in italics or make it a different color

I think it already is in a different font from what I can tell, but a different color would work wonders.

he does use a different fond...however, it would be useful to make a side note for people who don't code about standard practices...courier is always used to designate what you would actually type

So far this whole example is disorganized and a bit hard to follow. Overall it needs to be a bit better explained. You should probably just choose a few points and elaborate on them, rather than just listing everything you can think of.

I really do not understand what this is talking about at all. it prints the first 15 lines of what

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 23rd 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
```

The notation of this example confuses me.

The 'head -15' and 'tail -15' refer to text that would be typed into a command prompt or terminal.

I think the argument syntax for this function may confuse people not familiar with command line or terminal commands. I think a sentence or two describing the syntax, or at least making it less confusing to others might help.

I think these small examples would be better suited within the points that precede it, or at least identify the principals they match up with.

From solving this problem, I would guess that search engines use this divide and conquer method in their algorithms.

This is a really good problem, we did similar problems in 6.033 at around the same time

A page and a half was spent describing the logisitics of UNIX commands and how to use them to answer this question. This section is supposed to be describing how divide and conquer was used when UNIX was designed, and instead my time is wasted with a description of how to write UNIX commands, which I already know how to do. I understand that not everyone who reads this knows UNIX and UNIX commands, but in that case, I don't think this is a good example if so much explanation is necessary.

I liked this example and I think the time spent to explain the pipe combinator was worth it. Now I can think of the lines connecting nodes of a tree as the pipe operator and the node itself as the filter.

I think instead of listing these long commands that non course 6exers dont know much about, you could do more examples of the "print" one. There are many many problems using lists that could be relevant to divide and conquer

what do you mean? print only 23 lines? or print the 23rd line in a larger series?

This is a really good example, easy to understand for people who haven't had programming experience before.

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 23rd 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
   ```

I am confused about this, wouldn't the tail -1 be printing the 24th line?

You should think of the line as saying "tail -1 of head -23" which would mean "print the last line of the 1st 23 lines. "

Agreed, it seems as though we're printing the first 23 lines then the final line of the entire file.

the idea is that 'tail -1' is run on the 'head -23' output. There isn't a 24th (or greater) line at that point.

He talks about feeding outputs to other functions later. That should clear it up.

We don't always multiply, right?

A lot of the time it's just dimensional analysis, so it is pretty much multiplying by different unities.

I don't like the way this is introduced. I think it would be better to suggest it is a metaphor. Something like: "you can think as the filters as a pipe."

I don't like the way this is introduced. I think it would be better to suggest it is a metaphor. Something like: "you can think as the filters as a pipe."

I don't like the way this is introduced. I think it would be better to suggest it is a metaphor. Something like "you can think as the filters as a pipe."

this seems very intuitive to me. it was already explained above

I don't like the way this is introduced. I think it would be better to suggest it is a metaphor. Something like: "you can think as the filters as a pipe."

I don't like the way this is introduced. I think it would be better to suggest it is a metaphor. Something like "you can think as the filters as a pipe."

I don't like the way this is introduced. I think it would be better to suggest it is a metaphor. Something like "you can think as the filters as a pipe."

I thin

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 23rd 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
```

This section focuses a lot on programming and not much on divide and conquer. I'd like to see divide and conquer more incorporated in this section, otherwise I feel like I'm getting lost in programming and missing the whole point of this section.

I like the analogy

I feel like I'm missing what this has to do with estimation. It makes sense and is interesting and all of those good things, but I'm sort of wondering how it's relevant.

It's great that this part is so clear and easy to follow for people who don't understand UNIX, and I think it points out the slight inaccessibility of the previous section, when there were a lot of terms the UNIX-unsavvy couldn't follow.

As a before noted UNIX-unsavvy person, I agree. This is getting much clearer, although going over the principles in class would also help a lot. Please don't forget this is also registered as a course 2 class.

I agree. Sometimes I have trouble reading this book because I get distracted with unfamiliar concepts. But when you include definitions, the readings become clearer and I see how it relates to the problem at hand.

I agree with the comments from the others about this being well explained. I have no experience with UNIX at all, but this is nice and clear.

It's funny - in 6.033 we're reading the actual UNIX paper, and your writing is making it 10X more accessible.

I see how this models the tree and divide and conquer, but I don't see what we are going to be estimating

I thought that these printed in quantities of 15? or if not what was the 15 for

just curious, can you do head -24 | tail -2 and get the same result, or does the tail only print the LAST line of the input

I believe this would print line 23 AND line 24, and we were aiming to print only line 23.

As the previous reply says, you would get line 23 and 24 since you are saying the last TWO lines of the input.

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 23rd 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
```

For me this is where the divide and conquer analogy gets a little lost, although after thinking about it I suppose you could say defining the input is just another level of the tree. You might want to make that connection clearer.

good to know

So does this notation just mean that the output of head -23 is used as the input of tail -1?

Yes, thus the pipeline analogy.

So this will print only the 23rd line right?

therefore reads lines typed at the keyboard, prints the 23rd 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 23rd 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 [26]created by the computer scientist Donald Knuth, whose many accomplishments include the TEX typesetting system used to produce this book.

Comments on page 4

What does it mean that it reads lines from the keyboard? does this mean that after you type 23 lines of text it will execute the command?

I don't know much about programming and it's unclear what you mean by it reads lines typed at the keyboard. How does this differ from what we're looking for?

At the beginning of the problem we said we wanted the 23rd line of the file, so we probably want our filters to start with the contents of the file. Otherwise it would take whatever you are typing at the keyboard and return the 23rd line of that.

I'm not sure what the backslashes do

What is different between this and #2?

The difference between 2 and 3 clearly stated would be helpful.

I'm unclear on this. Is head going to read lines from the keyboard and then switch to the file? How does it know when to stop taking input from the keyboard?

What's the point of this if you can read from a file?

I'm not sure I fully understand this either. So does this put the input into a file, then read it from the file? So you're not directly reading the user input?

yeah i dont understand the nuances of this either

The wording is very technical and could use some more basic explanations about what it means to connect a file to an input.

While describing these three possible input methods for getting information into the pipe, I dont see what it adds to the divide and conquer section. It almost seems to be getting bogged down in the implementation of a specific case rather than the idea and effectiveness of divide and conquer.

I had the same comment while reading it. Although as a course-6-er I was interested in the explanation, it added nothing to the divide and conquer idea and obfuscated the reading.

therefore reads lines typed at the keyboard, prints the 23rd 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 23rd 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 [26]created by the computer scientist Donald Knuth, whose many accomplishments include the TEX typesetting system used to produce this book.

How is this different from 2? Just that you're "tricking it"? Is this in any way better than 2 or more useful in some other circumstance than 2?

By the same token, does it matter that the program now thinks its drawing from the keyboard input? Why would drawing from a pre-written file be any different for the computer, once you specify the file location?

I agree with these comments - I fail to see how this option is useful. Why would you ever want to "trick" the keyboard?

Sorry by keyboard I meant trick the system into believing it is receiving input from the keyboard.

I think it allows you to type in the whole line head -23<... the first time and from then on you can just type head -23 and leave out the file path. From then on the computer will assume that it should print from that path anyway.

I think, more importantly, the fact that we're being distracted by this question means that some of the detail here may be unnecessary to make the overall point, and seems to actually detract from our understanding.

I don't really understand what this is saying.

A minor point, but this should be "it's"

therefore reads lines typed at the keyboard, prints the 23rd 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 23rd 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 [26]created by the computer scientist Donald Knuth, whose many accomplishments include the TeX typesetting system used to produce this book.

what is the point of all of this? the text should be more focused on the "estimation" techniques rather then get bogged down in the programming.

I think the section is trying to drive home the idea that unix programs illustrate divide&conquer techniques, but I agree it seems too concerned with the details of these programs.

I also agree. It's good to see how these topics are used in real examples, but this seems a little much.

I also agree with this. I am finding myself just trying to understand how the programs work with the filters and such and forgetting I am even supposed to think about divide and conquer.

Yeah. The relation seems fuzzy and unclear. I think the simplicity of divide and conquer needs to be used in this example. It seems way to focused on details. Details that drive the reader away from the original intent of the example

I agree. I actually learned some useful things about Unix from here, but that's not what we're here for.

what do you mean by redirection operator, do you refer to < ?

I think that just means you're reading the file from standard input rather than straight from the file.

Yes, you are right, but it would be nice if < was called that above when it is first mentioned.

In what way is this slightly less efficient? I see that it probably takes longer but is that where the inefficiency lies?

Does this actually create a phantom copy?

ooh this sounds cool. Is it weird that I'm excited to read on to figure out how to solve this?

This example helps to clarify some of the ideas somewhat, but this whole section is still largely incomprehensible to me.

Thanks for the example. At first, I thought this might be from z to a, but now I get that its based on last letter of the words.

therefore reads lines typed at the keyboard, prints the 23rd 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 23rd 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 [26]created by the computer scientist Donald Knuth, whose many accomplishments include the TEX typesetting system used to produce this book.

what is this

It's a 'typesetting' program. You may have heard of LaTeX before; it's commonly used in the scientific environment to format and print formulas which would otherwise be cumbersome to use in word processing.

how is this different from other typing programs?

Yeah, I would like a short description of what this system does and why it's unique.

googling TeX will explain the difference:

http://latexforhumans.wordpress.com/2008/10/07/why-use-latex/

or

http://ricardo.ecn.wfu.edu/ cottrell/wp.html

latex is really great and i love it. however, i don't think the details of how it works re needed here, in this book. it's more of a point in passing, to show how awesome donald knuth is.

I agree. It's just a little aside to embellish the achievements of Mr. Knuth. It doesn't really matter what TeX is (one can always Google it) for this material, and as Unix Rule #1 says "Small is beautiful". No need to clutter the book with random stuff.

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:

Comments on page 5

Why/how does reverse alphabetizing the dictionary make this program better/more efficient?

I don't think this has to do with efficiency but is part of the problem definition/solution.

isn't this not really applicable to the divide and conquer idea- because its too hard to do this and where is the divide part

If I were asked to solve this problem on my own, I don't think I would come up with this as something I needed to do. I think it would help to explain why this is necessary.

it seems more natural to me to have these simply be the first 3 steps, for 4 steps total.

It is clear to me here that divide and conquer is being used in the problem but I think it should be stated at some point, probably earlier then this. This is the first time I realized we were breaking up the programming the same way we do problems.

Or maybe drawing a divide and conquer tree would be a useful way to show that?

Don't we want to get the next word after 'trivia' ending in -a in alphabetical order? But this seems like it would find the alphabetical ordered word after 'aivirt'

This was really helpful in that I could visualize the tree before we went through the problem, it would have also helped to have seen the second problem's 2 divisions verbally explained when it's branch was approached.

I understood what was being said here, but it was actually quite confusing the first time. Perhaps "reverse each word of a regular dictionary" would be more clear?

I agree with the replacing line with word, or simple clarifying we are talking about a unix dictionary where each line is a word.

I actually think this is a very helpful way to explain it for us to understand it.

This is sounding very much like an algorithmic breakdown. Interesting how algorithmic principle still prove useful in approximations.

I think it would be ususel in the class to go over some sorting algorithms, which was the first think that I think about with divide and conquer (merge sort etc) It's good for general knowledge and interview questions as well

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:

What is meant by this? Is this just saying that there is one command that might do this for us but for the purposes of this argument we are using divide and conquer?

I know its in a slightly different font, but it's still hard to notice that 'man' and earlier 'cat' are actual functions

isnt this the same thing as going through the toolbox looking for an operation? you just know that it exists instead of looking for it.

it would be helpful if you can actually provide some examples here

This is a very useful section.

Yeah that would make this easier to follow.

Like many people have pointed out, I feel like I understand the outline given at the top of the page, and how that uses the divide and conquer method, but I'm not experienced with coding or UNIX, so all this seems very foreign to me, and the lack of examples makes it hard to follow

I feel the same way. I understand the idea behind divide-and-conquer, the example goes from a very understandable one (printing the 23rd line) to this, and I had trouble following the code and UNIX commands. I agree that more examples that build up to this problem would be beneficial in grasping the material.

This is the sort of question that would benefit from expanding the tree to solve it.

I'd say reverse each line. then sort. then reverse again

```
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.

Comments on page 6

When these pipelines are added into the text, it would be helpful if each word and operation was pointed out and explaining the transition and what gets done.

Again, why exactly do we need this? We could (from what I understood earlier) get the same output without using it, only the program would terminate at the end.

nice I was right

Is the 3 by itself is enough to have it print the 3 lines?

I've never been a unix user but after reading this chapter I am inclined to start, the pipeline system and general intuitive open source options are incredibly appealing.

Does it really make sense to have stored these now oddly-ordered words in the dictionary on the computer, rather than in a new file?

I don't understand why we are learning all this UNIX terminology. Don't we just want the concepts?

good to know but it seems like you're using this when you already know what you're looking for. Maybe you just use this to check it's there. Why not just say like open trivia or whatever you're trying to do since you know the exact name

Is this the loose equivalent of quotation marks around a phrase in a google search?

Yeah, but with the added requirement of 'trivia' being an entire line by itself.

What would of happened if there were no words similar to trivia that were next to it, it seems to me that this method would not have worked. Or maybe I don't understand what is going on here.

```
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.

This attribute of grep seems almost too handy- is there no way if getting the position of trivia within this list of words?

It seems that those unfamiliar with UNIX would have a tough time figuring out all these commands - is there another way to do this?

I agree–the grep method and this -A option seem to basically perform the task for us, just in this one seemingly per-built in method(?). So aren't we just solving our problem using a method someone else already wrote to solve this problem?

I don't know UNIX and cannot understand if the -A option has to do with the fact that trivia has an A as the first letter when written backwards or if it is a general operation.

```
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 char-
acter
```

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

wait but aren't we sorting by the last letter? how is this helping us?

Have you not rearranged it yet by this point? Shouldn't all the words near trivia end in a if its reverse-alphabetized?

what is the point of this- I do not understand what I am supposed to get out of this that is useful

your previous line of code above had a "3" here instead of a "1", what just happened?

the previous 3 was so he could see the next 3 words. Now that he just wants 1 word he replaces the 3 with a 1.

then why would he write | tail -1 also? isn't that redundant?

Because the grep would return "trivia, alluvia", and he only wants 'alluvia'

Because the grep would return "trivia, alluvia", and he only wants 'alluvia'

very helpful, thank you!

I am having trouble following this code example.

Ok, so you are NOT actually saving your newly sorted list outside of the program, so to find the one after some other word (such as "dog") you would have to re-run the program

This is pretty easy to follow as I am doing it. This note also clarifies the confusion I had when I was trying to do the homework before reading this.

what does this means?

I don't really see why it matters what the computer says. It seemed as though everything before this worked and the extra steps taken don't really have to do with divide and conquer. Maybe all the technical terms are getting to me...

Again, why is this stuff relevant to this class?

where are these programs coming from?

so with one line of code you fixed the problem?

```
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 char-
acter
```

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

explanation would be helpful here...not just another computer jargon

So any words with accents would be deleted? Wouldn't this potentially change the solution if the word after trivia happened to have accents or uppercase letters

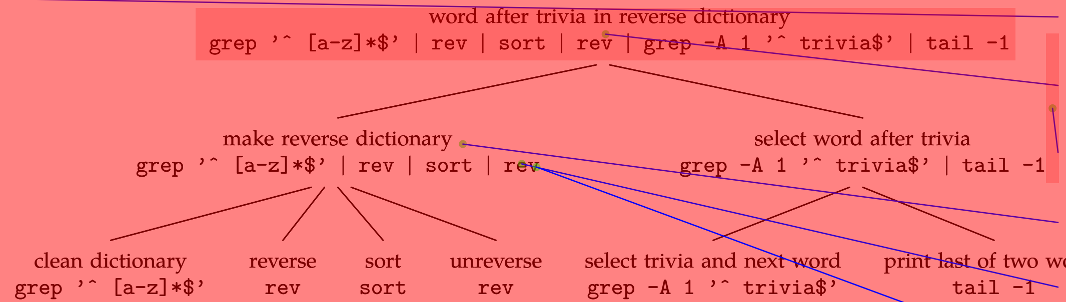
Yeah I'm not too happy with this compromise either, but I'm assuming there's no other simple way to do it...

Some of these code examples are hard to follow.

```
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 1.6 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.

I think by this stage in the chapter I got a bit lost...usually seeing a bunch of code frightens people. The chapter should go a little slower and maybe try and use simple pseudocode examples instead of focusing on UNIX.

I think I could've gotten the same out of the chapter without knowing the exact code. Could've used more reader friendly names for methods, but I do enjoy that I learned the something real

This is pretty simple to understand.

is this the code that reverses the dictionary?

tree takes a bit to understand, but makes sense

These trees with code in them are very difficult for me to understand.

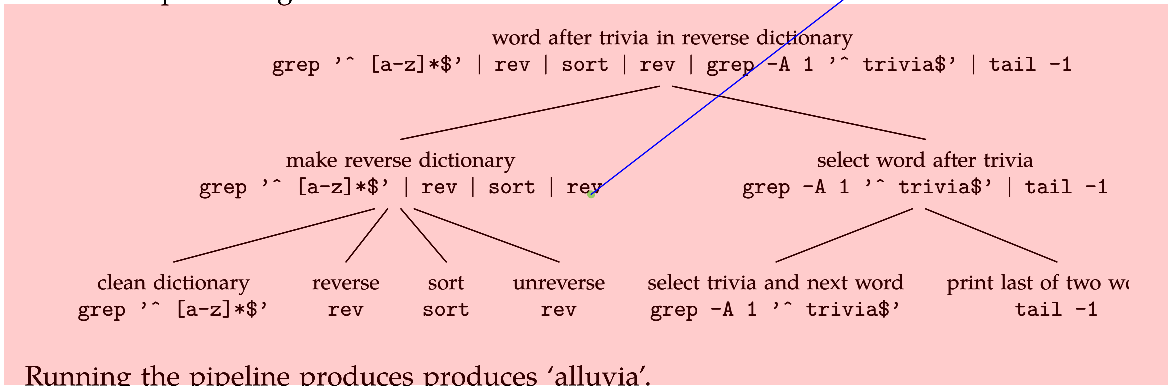
this reminds me a lot of 6.001

This tree is hard to follow by itself.

```
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 1.6 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.

I really like this diagram of a tree explaining the whole pipeline. It really ties back what we know of trees and divide and conquer to the UNIX example.

Although this tree certainly helps, a lot of this reading was very confusing to me as someone with extremely limited coding experience. I was able to follow the UNIX principles and even the basic head and tail topics, but after that I was pretty lost. Using this tree I was able to go back and understand a little bit more, but I am certainly not completely comfortable with the material.

And I guess to voice the opposite view point, this was an extremely enlightening example as it provided an interesting way for me to consider a problem I'd otherwise think of in a very informal fashion if I had to solve it, and probably take a lot longer to figure out, even with the experiences using Unix/Linux I have. So Thanks!

It would have helped me to see this, or at least parts of this, as we went through the problem and not the whole thing at the end.

I totally agree. Until I saw this tree I had totally lost the divide and conquer to trying to understand UNIX.

Although placing the tree at the end was helpful it would have been easier to understand if less UNIX coding was used and more english and logic was used.

Same here, I am not very familiar with UNIX but this tree really helped me figure out what was going on. It may be helpful to show this in the beginning as well

Maybe the best solution is instead to show the tree being developed as it progresses. Like when we realize we have to clean the dictionary, we can see that part added to the tree - the problem as described at the beginning of the section has a simpler tree that we add to so why not show the initial state and the complications added to it?

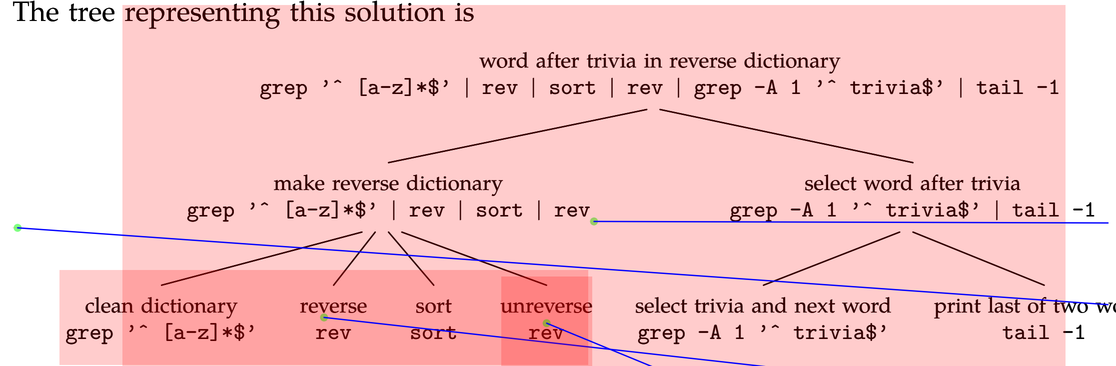
I agree, starting with a basic tree which was then improved upon would keep me focused on the divide and conquer concept as opposed to the syntax of UNIX

I strongly second the recommendation to slowly build the tree. I'm a relative noob in programming, so it was a bit difficult to follow the functions (since it's an entirely new syntax to me). However, if you could show the tree as it went along and then have it here as a cumulative one, that would be awesome.

```
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 1.6 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.

I agree. This tree is very helpful. Some of the above text is hard to follow, but this clears it up nicely.

Yeah I definitely agree too. I thought the sections about were long and somewhat difficult to follow, if someone doesn't have very much UNIX experience. This tree diagram, is a great way to represent the previous steps visually and to tie it back to the divide&conquer method we learned about in the previous sections.

I also like this tree, and feel that perhaps it(or a smaller version) would be helpful to have it earlier, because I began to jot a few words down in order to follow the logic

Yeah this really sums up everything well. While I don't quite understand all the code, this tree has helped me understand the divide and conquer application.

Yeah this is super helpful. It clarifies the whole process.

This tree really helps understand how the problem is solved- maybe it would have been better to put earlier since this explanation is so long

these aren't independent calculations. each one should be performed after the other, so they shouldnt be represented in this leaf format.

Aye, I agree. Are we supposed to assume that we apply these steps from left to right?

I'm confused about why this goes in the bottom part of the tree. Don't we want to select the word after trivia then unreverse it?

It would be helpful to see how exactly we got to alluvia from the tree.

I feel like it has been mostly described, but maybe seeing a short list of the two words proceeding and following trivia in the reverse dictionary would give a nice visual to wrap up the problem.

We would need to write a program to answer this, right? If so, then how does approximation come into this?

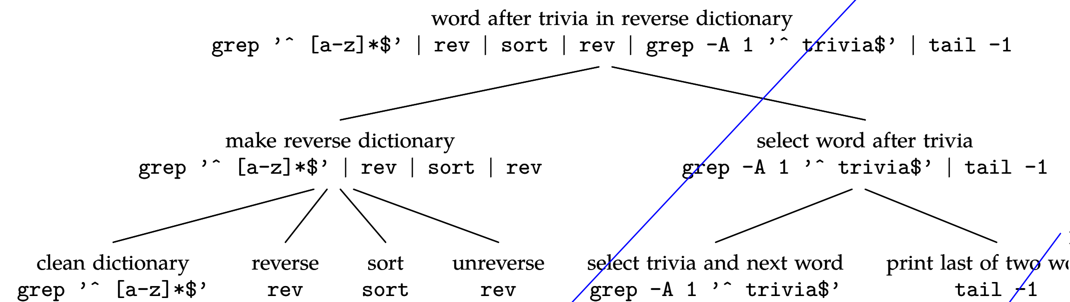
Can a person who is unfamiliar with Unix answer this question? I have no idea how I would begin (though I do understand the previous example).

...or any form of problem solving

```
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 1.6 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.

I feel like I understood divide and conquer okay before this chapter, but as someone who doesn't know anything about UNIX, this chapter was a hard read and I didn't get much out of it in the end...

I think he's just trying to make the point that divide and conquer is a common, useful tool for solving problems even when used outside of the confines of estimation. However, it would probably help if this was stated at the beginning of the chapter.

I agree, this helped me to realize that divide and conquer is used in other instances but the technical pieces of UNIX are unfamiliar to me and made the section harder to understand.

I also agree. I think divide and conque is something people generally do, especially when things are exact like computer code. I'd like more talk about how to go about breaking things down. Hints, tips, etc..

it wouldve been better to leave it short.

Agreed, I got so lost in the programming discussion that when we came back to divide and conquer I felt like I had missed the whole point.

Definitely agreed. I am even in course 6 and I understood all the details about UNIX and programming, and I still thought this example was excessive. There was way too much explanation necessary for such a simple concept as divide and conquer. And no where throughout this example did I even remember that this was supposed to be about divide and conquer. I would delete this whole example from the book.

I think we can find a better example to use to show how common divide-and-conquer reasoning is.

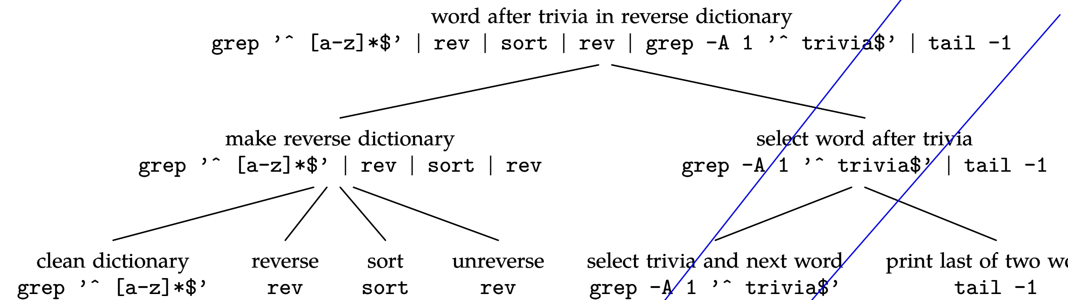
A very good example of divide and conquer, but the result is very discrete and precise: not an approximation or an estimation.

I think the long explanation of various unix functionalities didn't add to the explanation and instead made it all significantly more confusing, even though I understand how to use most of them.

```
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 1.6 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.

The UNIX Part of this doesn't quite help the final message; I feel like it's rather off topic from Divide and Conquer. It's cool to learn, and I understand what it's trying to prove, but I just don't get the helpfulness to divide and conquer. Maybe after lecture it will become more clear.

Is the UNIX example relevant specifically? Would other programs be just as suitable as an example as long as they are modular in design?

I'm still not sure I understand until where I expand the tree. Is there a rule of thumb about where to stop expanding?

I agree. In this case it looks like the author expanded the tree until he could use single methods to solve each problem (without combining methods with the |). But I'm not experienced with UNIX or coding so I'm not sure...

You should keep going until you have numbers that are manageable and are likely to be correct (at least to the order of magnitude) or numbers that you know are correct. Typically, these numbers are much smaller than the actual answer. In a way, we're approximating a big thing with many small things that combine together.

More generally, expand until each problem (leaf) is soluble on its own. This may vary for different people. Just think of how you carry out any task that involves more than one step or piece.

was this explicitly stated before? it makes sense though.

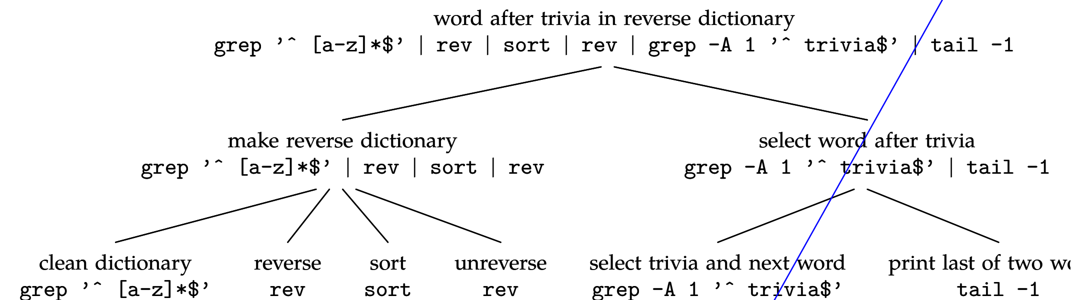
This is true for most divide and conquer cases where you're trying to estimate but it doesn't seem applicable here. Breaking down the problem makes it easier to realize how to code for an answer, but since you're relying on a computer program to find the answer, accuracy doesn't seem to be a concern. This point was much clearer in the economics example.

I really think divide and conquer is the easiest way to solve complicated problems. Also, knowing how to break the problem into steps is extremely helpful, but I find that to be the most difficult part.

```
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 1.6 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.

does this actually ensure accuracy? it seems like the more you subdivide, the more you need to multiply guessed numbers together, possibly getting a much larger answer than expected.

I know someone brought this up on the earlier section about trees, but does the error multiply similarly up the tree? What is the relationship between the size of the error and the number of leaves? (I doubt its a constant for all types of problems but it would still be interesting to study)

I feel like this is just a reflexive fear. Everybody seems to keep fearing the possibility of errors compounding and snowballing out of control, but from something as simple as the health cost example in class, I've found that if you keep trying approaches to a problem until you reach an angle where you're comfortable with the numbers, the numbers won't be too radically off.

Just an observation.

Well I like the point is to break the problem down into things you can estimate with accuracy and get rid of that error.

It seems that trees also represent nearly all other kinds of problem solving and logical processes. Engineering system design, analytical thinking, etc.

That's an interesting thought. I agree it is pretty useful, but I'm wondering about the efficiency of using trees over other methods.

There are other diagramming options for other types of problems like flow charts, organizational charts, Gantt... Trees are useful but not universally, as we've seen with our difficulties in illustrating connections and redundancy.

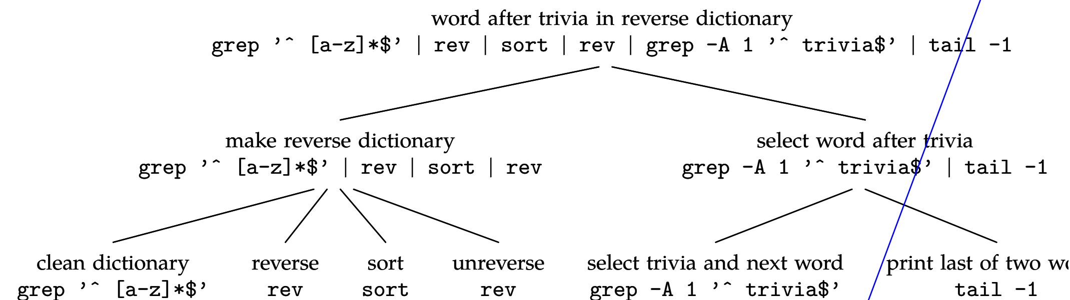
And in books, like this chapter... I'd still prefer something different than the text processing example, but maybe that's just me.

I think it's a good idea to have some example that's not computing numbers, although it doesn't have to be a text processing. But it might be this is the simplest example to read without Linux experience and potentially grasp the first time.

```
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 1.6 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.

Could you possibly provide an example of how divide-and-conquer reasoning is used in economics? I think it would be helpful for future reading of this section, but I would also be interested in learning about an example currently.

I agree, an example would be nice. Otherwise this point feels I little out of place in the conclusion, given we haven't seen such an example before.

The oil problem is sort of economics...ish. More so than the others.

It seems that by saying "and even economics," that divide and conquer is used for only the problems mentioned, but I think you mean to say that it applies to many problems in life, sometime obvious and sometimes not.

I think by saying even economics, the author is trying to say divide-and-conquer's usefulness expands more than just the examples mentioned and suggests a possible field you hadn't thought of.

I don't think we need an example of that. Especially not at this part of the text. There are already some stellar examples of how this works and we don't actually need an economics specific one to udnerstand it better. This bullet point just serves to illustrate how wide-reaching this technique is beyond just estimation (plus it wouldn't really fit here... I mean, it's a bullet point).

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 1.7 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 1.8 747
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 1.9 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 1.10 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 1.11 Bandwidth
Estimate the bandwidth (bits/s) of a 747 crossing the Atlantic filled with CDROM's.

Comments on page 9

So the point of this was to help us organize big problems, not estimate. Ah.

copy edit: I think there should be a comma between cousin and abstraction, since cousin is the object, not an adjective. Abstraction is then in apposition to cousin.

Is there a constant for the density of air that we should know?

Is there a section of the book with useful constants? Maybe a page or a couple pages like the one in our desert island pretest? That would be a pretty nifty trivia cheat sheet.

OCW Spring 2008 for this class has a version of the book, and page 2 is Back-of-the-envelope numbers, basically the same as the desert island sheet. You can start making your own based on these, but please don't beat me at trivia with it.

For the record, will this book give solutions to these "try-for-yourself" type questions?

It should in the ideal world. But I suspect that, in order to actually finish the book, I may just have to release it into the world without them. But it will be a freely licensed book, and I hope that people will contribute solutions...

Who moves a full fish tank bigger than a few gallons?

Problem 1.12 Repainting MIT
Estimate the cost to repaint all indoor walls in the main MIT classroom buildings. [with thanks to D. Zurovcik]

Problem 1.13 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 any handy UNIX or GNU/Linux system.]

Problem 1.14 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 **tr**anslate any non-alphabetic character into a newline. Useful utilities include `tr` and `uniq`.]

Nice, I really like this question, but it should be more general when it's published.

My UNIX underlying OS X does not have a tac man entry or allow me to run that program.