What Makes a Well-Defined Algorithmic Problem?

The challenge of problem formulation in computer science

Transforming a (typically imprecise) scientific problem into a well-defined algorithmic problem is an often under-appreciated challenge in applications of computer science such as the analysis of social networks or computational biology. Some biologists assume that as soon as a biological problem is stated, its transformation into a computational problem will be automatically taken care of by bioinformaticians. Try to transform the biological problem "How does the cell work?" into a computational one, and you will see that this task is far from being simple! Moreover, some biologists are not trained to distinguish an ill-defined algorithmic problem from a well-defined one that precisely describes the input along with an objective function specifying what output will solve this problem.

An example of an applied computer science problem

(The following is taken from *Bioinformatics Algorithms*, by Compeau & Pevzner: http://bioinformaticsalgorithms.org).

Imagine that we stack a hundred identical copies of a single edition of the *New York Times* on a pile of dynamite, and then we light the fuse (Figure 1). We ask you to further suspend your disbelief and assume that the newspapers are not all incinerated but instead explode into smoldering pieces of confetti. How could we use the tiny snippets of newspaper to figure out what the news was?

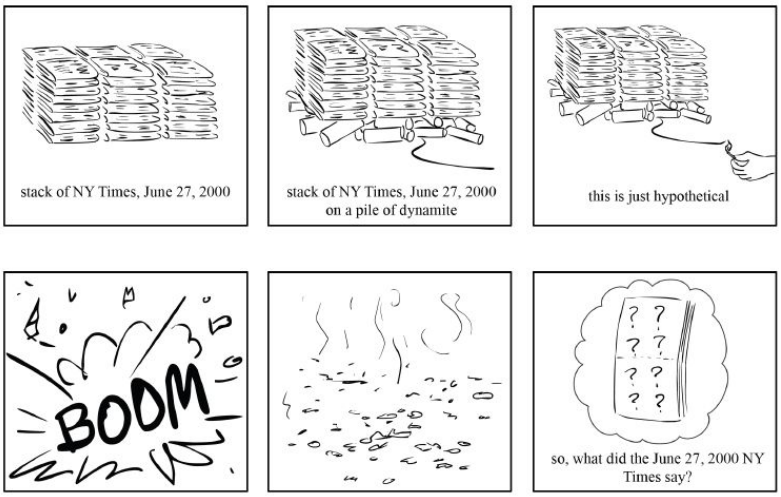

Figure 1: Don't try this at home! Crazy as it may seem, the Newspaper Problem serves as an analogy for the computational framework of genome assembly.

Because we had multiple copies of the same edition of the newspaper, we need to use overlapping fragments from different copies of the newspaper to reconstruct the day's news, as shown in Figure 2.

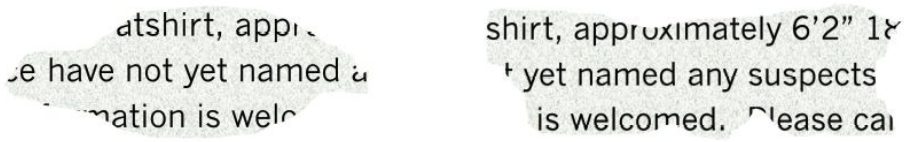

Figure 2: Using overlapping shreds of paper to figure out the day's news.

Exploding newspapers offer a good analogy for **genome sequencing**, determining the order of nucleotides in a genome. Biologists still lack the technology to read the nucleotides of a genome from beginning to end in the same way that you would read a book. The best they can do is to sequence much shorter DNA fragments called **reads**.

The traditional method for sequencing genomes is illustrated in Figure 3. Researchers take a small blood sample containing millions of cells with identical DNA, use biochemical methods to break the DNA into fragments, and then sequence these fragments to produce reads. The difficulty is that researchers do not know where in the genome these reads came from, so they must use overlapping reads to reconstruct the genome. Thus, putting a genome back together from its reads, or **genome assembly**, is just like the newspaper problem.

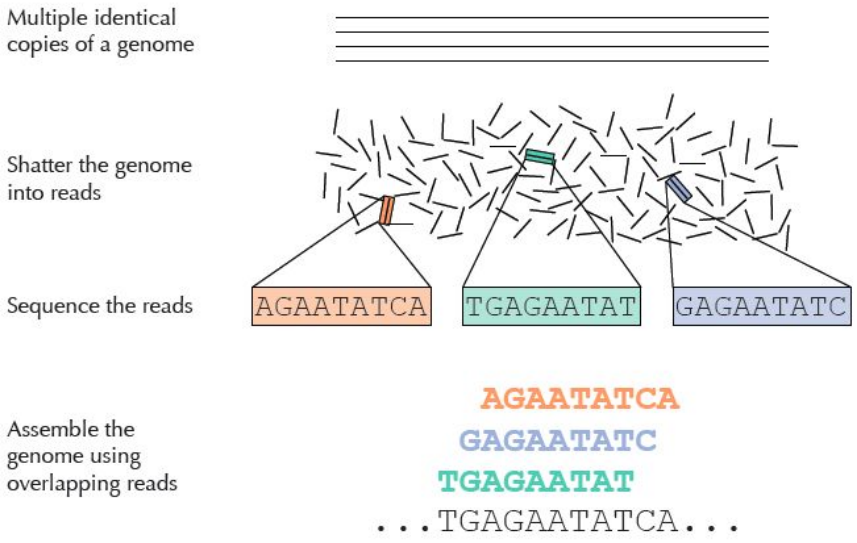

Figure 3: Sequencing genomes by breaking them into reads, identifying the sequence of nucleotides in each read, and assembling overlapping reads to sequence the genome.

At this point, you are probably wondering what is the *exact* algorithmic problem we want you to solve. Usually in algorithmic courses, you are facing a

well-defined algorithmic problem that the instructors have already formulated for you. Now, you have to take initiative in your hands and start formulating the problem yourself!

If you succeeded in problem formulation, Alice will solve it!

Imagine that you have paid a million dollars to a brilliant computer scientist to answer, "How do I assemble a genome?" The only catch is that Alice has not studied biology since high school and only knows computer science. If you describe your problem to Alice a page with biological jargon, then your problem will go unsolved; it is up to you as a computational biologist to translate the biological problem into a well-defined algorithmic one.

Your algorithimic problem may have multiple correct solutions, and Alice is free to report any of these solutions. In fact, she would prefer to report an easy-to-find solution to collect the prize! This is why it is important that your problem is formulated in such a way that each solution returned by Alice will answer the question you are really interested in.

Ten simple rules for problem formulation

Here are a few rules you should pay attention to when trying to devise algorithmic problem formulations.

1. Be specific and precise when describing the problem's objective, constraints, input, and output.
2. The problem formulation should be short and elegant. If you can shorten your problem formulation, do it! If, after introducing all definitions, your formulation exceeds a paragraph, then you are probably on the wrong track.
3. A well-formulated algorithmic problem specifies how to determine whether the output is correct given the input. If it does not, how would you be able to check that Alice's solution is correct?
4. If Alice returns an answer that doesn't have any ultimate biological relevance, then you have wasted a million dollars!
5. An algorithmic problem has nothing to do with the subject area – try to avoid terms like "reads" or "genome." Two completely different real-world problems (such as assembling genome or finding an optimal route from New York to San Francisco) may end up in identical problem formulations. For example, instead of referring to a "genome", refer to a string of symbols. If you insist on using biological terms in your input/output description, then you should first define them in purely mathematical terms.
6. All definitions of new terms should be stated before the problem input, and there should not be any text between the input and output. Do not

introduce definitions inside the input and output.

7. If a certain variable shows up in the input, then it should be used to produce the output.
8. Problem formulation is independent of their complexity; even intractable problems can still be well-formulated.
9. Algorithmic problem formulation is not an algorithm – do not provide an algorithm in your problem formulation! Many students find themselves trying to design an algorithm before formulating an algorithmic problem, which only causes problems.
10. The problem formulation should not get into the details of the format of the data representation. For example, instead of saying "a number N in the first line, and N strings in the second line," you can simply say "A set of N strings."

Examples of ill-defined problems

Below are some examples of ill-formulated algorithmic problems modeling the biological problem of genome assembly – try to figure out what is wrong with each one.

Problem Formulation 1
Input: fragments of a genome sequence
Output: the complete genome sequence

Is the genome given? If not, then you should write instead:

Input: fragments of the sequence of an (unknown) genome

However, we want to avoid all biological references here. Using the term "genome" makes this a vaguely defined algorithmic problem.

Formulation 2
Input: A set of reads.
Output: Construct an overlap graph with vertices as reads and directed edges representing overlapping reads. The length of each edge in the graph as the read length minus the length of the shared suffix and prefix. Find a Hamiltonian path in the graph, which indicates a candidate genome.

Algorithmic problem formulation is not an algorithm - do not provide an algorithm in your problem formulation! However, the following is the well-defined problem formulation if you rigorously define the concept of an overlap graph before stating the problem.

Input: A set of strings.

Output: A Hamiltonian path in the overlap graph of these strings.

To assemble a genome means to put together many of its reads obtained experimentally. To reconstruct the genome sequence, we should overlap prefixes and suffixes of many reads, given that these reads are small pieces of the complete genome. If you can shorten your problem formulation, do it!

Formulation 3
Input: in the first line, a number *N* representing the amount of reads; in the next lines, *N* strings of variable length drawn from the finite set {A, C, G, T}.
Output: a string representing the complete genome. The beginning of the sequence may vary if the resulting string is circular.

This problem formulation suffers two issues. First, it should not get into the details of the format of the data representation in the input. Second, what does it mean for a string to "represent a complete genome"? How does Alice know which strings represent a complete genome? This formulation suffers the same problem as Formulation 1, in which Alice can simply output a concatenate of input strings.

Formulation 4
Input: A series of 'reads' -- genome fragments of known length and structure. These are represented as strings with elements from the character set {A, C, G, T}. There is a minimum length *k* of the set of reads.
Output: Genome sequence -- a string with elements from {A, C, G, T} -- in which each read appears at least once as a substring. Read sequences can overlap within the genome and the genome should be a minimal (e.g. shortest) such sequence.

Here, if the last sentence of the input doesn't affect the output, then it should be removed. By assumption, each read is a subsection of the complete genome. The set of reads may have come from more than one copy of this genome. Every substring of the genome of length *k* appears in some read.

Furthermore, all definitions of new terms should be stated before the input, and there should not be any text between the input and the output. Definitions (like "read sequences can overlap") should appear before stating the problem, for example.

Formulation 5
Input: fragments of an unknown string *S*.
Output: the string *S*.

This is a short, snappy problem statement, but Alice can pass any concatenation

of input fragments as the output, which is not what you want. For a simple example, you could give Alice the fragments GCC, ATG, and TGC. The answer that you would like is ATGCC, and yet according to the above formulation she could simply return GCCATGTGC.

In the next attempt, we get more precise by quantifying fragments as substrings.

Formulation 6
Input: A collection of strings composed of the symbols {A, C, G, T}.
Output: A string whose substrings are equal to the input strings.

Yet Alice could still respond back, "Substrings of what length?" For example, ACTG contains the substrings A, C, T, G, AC, CT, TG, ACT, CTG, and ACTG. Since you have not specified which substrings you are interested in, say that you passed Alice the input {AG, GT, TC}, and she could reply that there is no string with the substrings {AG, GT, TC}, and rightly collect her million dollars.

We can do better if we instead demand that the string should have minimum length.

Formulation 7
Input: A collection of N strings S (reads) formed from the alphabet {A,C,G,T}.
Output: A string of minimum length that contains all strings in S as substrings.

If you can shorten your problem formulation, do it! In this case, why do we need a variable N?

Towards a well-defined Genome Assembly Problem

After seven problem formulations, you are hopefully ready to formulate a well-defined algorithmic problem modeling genome assembly.

Formulation 8
Input: A set of strings S.
Output: A shortest string that contains all strings from S as substrings.

STOP and Think: There is still one additional error with the above problem formulation. What is wrong?