PART I: Algorithms (mostly strings)

Mini-forward

These texts were written for "Introduction to Computational Biology and Biological Modeling" course I taught for 16 years at Penn. The text is somewhat wordy. I've put things that are technical or side discussions in boxes.

I wrote this mostly to collate the topics I wanted to cover and I apologize that I am missing explicit references to other papers or textbooks from which I sourced my material. It is inexcusable sloppiness but I haven't had the time to rewrite everything for more public distribution except for the students who take my course (who are also given a list of important books for reference). Some of the books and references I draw heavily from are:

Algorithms on strings, trees, and sequences: computer science and computational biology, 1997. Dan Gusfield, Cambridge Univ Press, New York, NY.

Computational Molecular Biology: An Algorithmic Approach, 2000. Pavel A Pevzner, MIT Press, Cambridge MA.

Biological sequence analysis: Probabilistic models of proteins and nucleic acids, 1998. Richard Durbin, Sean R. Eddy, Anders Korgh, Graeme Mitchison, Cambridge Univ Press, New York, NY.

Statistical Learning Theory, 1998. Vladmir N. Vapnik, Wiley, New York, NY.

If you happen to see this document in the wild, please accept my apologies and also if you see any errors, I would appreciate any corrections.

Unit 1: String search problems

Genomic data mostly consists of sequences of letters representing the nucleotides, A, C, G, and T for DNA (with U instead of T for RNA) and 20 letters of the amino-acids. Here are the codes for amino acids:

Amino Acid	Three letter code	Single letter code
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartic Acid	Asp	D
Cysteine	Cys	C
Glutamic Acid	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoluecine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Theronine	Thr	T
Tryptophan	Tryp	W
Tyrosine	Tyr	Y
Valine	Val	V
Asn or Asp	Asx	B
Glu or Gln	Glx	Z
Leu or Ile	Xle	J
Unknown	Xaa	X

There are a few codes at the bottom of the above table for ambiguous amino-acids (only those that are commonly difficult to distinguish by chemical methods). There are ambiguity codes for nucleotides as well:

Nucleotide	Code
A or G	R
C or T	Y
A or C	M
G or T	K
C or G	S
A or T	W
A or C or T	H
C or G or T	B
A or C or G	V
A or T or G	D
Any four	N

Typical genomic data is some string of amino-acid or nucleotides that is corresponds to biological units such as a "gene." This data is most commonly stored in a text file. The most common format for text files of nucleotides or amino-acids is called the FASTA format (invented by David Lipman and William Pearson for one of the first string analysis program packages). The format is simple. It has one line called the header that starts with the symbol ">". The header line has freeform information about the sequence that follows. The next set of lines contains the string of interest, which terminates either at the end of file or with the appearance of the next header line.

Here is the FASTA record (i.e., one string in FASTA format) for the Drosophila melanogaster olfactory receptor gene Or22a.

```
>gi|24580887|ref|NM_078729.2| Drosophila melanogaster odorant receptor 22a (Or22a), mRNA
ATGTTAAGCAAGTTTTTTCCCCACATAAAAGAAAAGCCATTGAGCGAGCGGGTTAAGTCCCGAGATGCCT
TCATTTACTTGGATCGGGTGATGTGGTCCTTTGGCTGGACAGAGCCTGAAAACAAAAGGTGGATCCTTCC
TTATAAACTGTGGTTAGCGTTCGTGAACATAGTAATGCTCATCCTTCTGCCGATCTCGATAAGCATCGAG
TACCTCCACCGATTTAAAACCTTCTCGGCGGGGGAGTTCCTTAGTTCCCTCGAGATTGGAGTCAACATGT
ACGGAAGCTCTTTTAAGTGCGCCTTCACCTTGATTGGATTCAAGAAAAGACAGGAAGCTAAGGTTTTACT
GGATCAGCTGGACAAGAGATGCCTTAGCGATAAGGAGAGGTCCACTGTTCATCGCTATGTCGCCATGGGA
AACTTTTTCGATATTTTGTATCACATTTTTTACTCCACCTTCGTGGTAATGAACTTCCCGTATTTTCTGC
TTGAGAGACGCCATGCTTGGCGCATGTACTTTCCATATATCGATTCCGACGAACAGTTTTACATCTCCAG
CATCGCCGAGTGTTTTCTGATGACGGAGGCCATCTACATGGATCTCTGTACGGACGTGTGTCCCTTGATC
TCCATGCTTATGGCTCGATGCCACATTAGCCTCCTGAAACAGCGACTGAGAAATCTCCGATCGAAGCCAG
GAAGGACCGAAGATGAGTACTTGGAGGAGCTCACCGAGTGCATTCGGGATCATCGATTGCTATTGGACTA
TGTTGACGCATTGCGACCCGTCTTTTCGGGAACCATTTTTGTGCAGTTCCTCCTGATCGGTACTGTACTG
GGTCTCTCAATGATAAATCTAATGTTCTTCTCGACATTTTGGACTGGTGTCGCCACTTGCCTTTTTATGT
TCGACGTGTCCATGGAGACGTTCCCCTTTTGCTATTTGTGCAACATGATTATCGATGACTGCCAGGAAAT
GTCCAATTGCCTCTTTCAATCGGACTGGACCTCTGCCGATCGTCGCTACAAATCCACGTTGGTATACTTT
CTTCACAATCTTCAGCAACCCATTACTCTCACGGCTGGTGGAGTGTTTCCTATTTCCATGCAAACAAATT
TGGCTATGGTGAAGCTGGCATTTTCTGTGGTTACGGTAATTAAGCAATTTAACTTGGCCGAAAGGTTTCA
ATAAGTTGAGAGGGACGAGCTCTGCTACTATTATATTATATATTATATTATATTATATATATATTATTTT
ATATTATATATTGCTGTACCCTAATAAATATTTA
```

One of the most basic things to do in handling genomic sequences is to look for particular strings of interest from a large collection, say the Genbank (http://www.ncbi.nlm.nih.gov/genbank/), which holds most of the genomic sequence information submitted by researchers all over the world. Or, perhaps look for a particular substring in a much larger string. We will call this the "Search Problem" or the "String Match Problem." There are many kinds of searches one might want to do, each with varying degree of difficulty. For each, we have to consider not only how to do it, but how to do it fast. A fundamental consideration in all genomics-type of problems is that the size of the problem is absolutely huge. Being able to solve a small problem can be very very different from having it work on, say the entire collection of webpages in the world. A mammalian genome has approximately $3x10^9$ bases of DNA information or ~4×10^8 bytes. A typical search will have to work over many thousands of such genomes.

We can approximately classify the kinds of searches we want to do as:

1. Exact match
2. Inexact match
3. Inexact match allowing for rearrangements
4. Abstract patterns
5. Database dependent searches

An ***"Exact Match***", is like what it sounds like. It is a search for an exact occurrence of a particular string. You can open up a MS Word document, for instance the class syllabus, and do a search for "Computational Biology". In this case, we may either want the first instance of the search string or perhaps all such instances. If you happened to type in "Cmputational Biology", you won't find anything. However, if you do a Google search for "Cmputational Biology", you will get a list of potential search terms that are close to "Cmputational Biology". Here, the Google algorithm carries out an "***Inexact Match***" search amongst the set of common search queries that it has stored. Inexact matching can have many different flavors. The simplest is where the difference between the target in the database and the query pattern is small and does not change the linear order of the symbols. A tool developed at NIH called BLAST does many types of inexact searches for databases maintained at NCBI (National Center for Biotechnology Information). However, once the difference becomes greater, the computational difficulty becomes much greater—for example, a google search for "Cmpttnl Blgy" will turn up some very different things.

More complex inexact match problems happen when we want matches to rearrangements of the query pattern such as permutations. For example, "Biology Computational" is a permutation of the original query. DNA strings can have such rearrangements due to experimental errors or through natural mutations. "Biology Computational" is an example of a circular permutation. (To obtain a circular permutation of a string "ABCDE" imagine we have a string with "ABCDE" written on it and then connect the string in a circle, then break it randomly in some place.) Possible circular permutation of "ABCDE" includes strings like "EABCD", "CDEAB", etc. Circular permutations can be an important type of biological rearrangement because a fragment

of DNA or RNA can circularize (e.g., plasmid) and then become linearized by a break at a different place. More complex rearrangements can be things like "ComputaBioltional logy" (transposition) or "lanoitatupmoC Biology" (inversion) or "CompppmoCutational Biology" (inverted duplication), etc. More commonly, we might want to search for **abstract patterns**; that is, patterns that we define by certain rules. For example, a simple kind of abstract pattern is one that allows multiple possible variations of a letter: "[C|K]omputational [B|V]iology" where by [C|K] we mean either "C" or "K". There are commonly used syntaxes to describe such abstract patterns. In the programming language Perl and the operating system LINUX/UNIX, one particular type of syntax constructions is called "Regular Expressions." We will learn more about them later, but you can get more information from the web. Some abstract pattern search can be quite complicated such as "palindromes of size 10." Many kinds of searches depend on the database; that is, the **search type is a function of the database** that you are searching through. Examples include: "Best Matching String", "Longest run of A's", "strings that appear more than 3 times", "the most frequent 5 letter word", etc. All of these search problems get progressively harder to solve and harder to carryout efficiently.

We now introduce some definitions and terminologies that will help make the discussion more precise:

Alphabet: A finite collection of symbols; e.g., {A, C, T, G}, {A-Z, 0-9}. A blank space, " ", is also considered a symbol as well are other punctuation marks. Sometimes we will make a distinction between types of symbols, using terms like "terminal symbols" and "non-terminal symbols" (more later). Unless there is a potential for confusion, in this text, I will use the term "symbol" and "letter" interchangeably.

Words, strings: A consecutive set of symbols from the alphabet set. In general, I will put quotes around a string, but sometimes the quotes may be dropped.

k-tuple, k-mer: Typically the term "k-tuple" is used to denote an ordered set of k things; e.g., a vector of numbers (1, 2,1, 3) would be called 4-tuple. A k-mer is typically used to denote an ordered set of symbols; i.e., it more explicitly refers to composition of symbols. In much of computational biology, these two terms are used interchangeably.

Most of the time, we might use a variable to denote a letter (symbol), a word, or a string. For example:

S = "ACCCT"; T = "A"; U = " ", etc.

We write, ST to denote string concatenations such that ST = "ACCCTA" and TS = "AACCCT". We also use a power notation, T^n, to denote multiple application of the concatenation operation so T^3 = "AAA". More uncommonly, the power notation might be applied to an alphabet. So, if D = {A, C, G, T} is the alphabet set for DNA, then D^4 might be used to denote the set of all 4-mers from the DNA alphabet.

Substring: A substring of a string, say "ACCCT", is some string that is a contiguous subset of the string. For example, "CCC" is a substring of "ACCCT". However, "ACCT" is not a substring of "ACCCT" since it is not contiguous. As in many standard mathematical definitions, the string is a substring of itself. That is "ACCCT" is a substring of "ACCCT".

Subsequence: A subsequence of a string, say "ACCCT", is any consecutive subset of the original string. For example, "ACCT" is a subsequence, but "CACCT" or "ACCTT" is not a subsequence. If a string is a subsequence of another string, we are able to align the two strings such that there is a correspondence between the symbols along with "gap symbols" in the substring for any missing letters. For example,

```
"ACCCT"
"ACC-T"
```
or
```
"ACCCT"
"A--CT"
```

or
"ACCCT"
"---CT"

Given a string S = "ACCCT" and so on, we will use S[1], S[2], etc. to denote the 1st, 2nd, symbol, etc. We will also use the notation S[1...3] or S[2..4] to denote the substring that goes from 1st to 3rd symbol or the 2nd to 4th symbol and so on.

Prefix and suffix: Given some string S of length n, a prefix is any substring that has the form S[1…k] for and $k \leq n$, and a suffix is any substring that has the form S[k..n] for any $k \leq n$.

Kleene star: There is a special notation called the Kleene star that represents the set of all strings that can be generated from some finite set of alphabet or strings by (countably) consecutive application of all possible concatenations. For example, if S = {A,T} then S* = {A, AA, AT, TA, AAT, ATAT, ….}.

There is a special term called " language", the consequence of which we will see later. Here is the definition.

Language: If A is some alphabet and A* is the Kleene star of A, then a language is some subset of A*. Different languages of the same alphabet are different subsets of A*.

This definition of a "language" is not exactly what we usually think of when we think of natural languages. The term comes from the study of formal languages (as opposed to natural languages) that we will discuss later. However, the term is motivated by natural languages. Consider the standard alphabet set used in English along with punctuation marks and maybe some non-standard symbols like ê. Let **A** = {A, B, C …} be this alphabet set, then **A*** represents all the possible (including nonsense) words and sentences that can be composed from this alphabet. English is a particular subset of **A*** that corresponds to the usual kinds of English words and sentences while French is a different subset of **A***. While this seems a bit overly abstract, precise notions like this can help modeling biological sequences. For example, a "gene" is comprised of sequences of nucleotides. Let **D** = {A, C, G, T} be the alphabet for DNA then **D*** is the set of all possible strings comprised of the four nucleotides. We know that things like "AAA…A" (10,000 A's) are not genes as far as we know, while say 10 A's in a row might be part of a gene. So a gene is a subset of **D***--we could even call it a *Language of The Gene*. The question is whether we might know which subset of **D*** belongs to the *Language of The Gene* and which do not belong. How do we know whether a particular string of letters do or do not belong in English? While not always exact, we think of a string of letters as being part of the English language if they conform to certain rules of grammar. One might think it would be nice if there were a grammar for the *Language of The Gene*. We will consider many of these issues later.

Unit 2: Exact string search

Now, back to the search problem. We will first tackle the exact search problem. Here is the problem:

Exact Search Problem: Let S be some string and P another string. If P is a substring of S, return the offset of the first match position. That is, if S is a string of length *n* and P is a string of length *m* and S[*k+1..k+m*] = P[*1..m*] (and $k+m \leq n$) then return *k*, where *k* is the smallest such number.

In real applications S might be a very large collection, sometimes an entire database such as the Genbank. (Note that we are returning *k* and the pattern we want starts from the *k+1* position; therefore, *k* is called the offset. We should also note that whether we return the offset or the starting position doesn't really matter. There is just a convention to return the offset.)

It doesn't take too much thought to see that there is a simple algorithm to solve this problem. Here is a pseudo-code. Pseudo code contains statements that outline the steps of an algorithm but typically do not correspond to an actual programming language.

```
i = 0
while (i <= n-m) {
        k = 1
        result = 1
        while (k <= m) {
                if(S[i+k] <> P[k]) then result = 0
                increment k
        }
        if result = 1 then return i
        increment i
}
return "not found"
```

Fig 1: Naïve Search

If you have never programmed before, this pseudo code will need a bit of explaining. We first set an index variable called **i** to zero. This index will go from zero to near the end of the S string and refer to positions within the string S. In fact, it will go up to the offset point of the last possible place that the string P can be found in S. For example, if S = "ACCT" and P = "CT", then $n = 4$, $m = 2$, and **i** will run from 0 to 2. The part of the code that starts with "**while (X) { }**" says we should repeat everything within the brackets "**{ }**" as long as the condition **X** is true. These kinds of constructions are called a "loop"; that is, keep looping through statements inside the brackets as long as the condition **X** is true.

We next set another index **k** to 1 and use it to refer to the positions within the string P. We also set another variable that we will call **result** to the value 1. This variable is what we call a "flag". The flag variable will tell us after some calculations whether we have found a match to the string P. The second **while** loop now runs from the first position of string P to the end of the string P and tests if the symbols of P are different from the symbols of the string S, starting from the **i+k** position of S. If anywhere along the string P, we find a mismatch with S, we set the flag, **result = 0**. Therefore, after we do this loop, **result** variable will stay at the value 1 if there is a substring in S that matches P, but if there is any symbol that doesn't match it gets set to a value of 0. If **result = 1** after we go through the inner **while { }** loop, we will have found the first match and we return the current value of **i**, which is the offset. Stating "return the value of **i**" also implies that we stop all calculations as soon as this happens—this is called "breaking out" of the various loops. Otherwise, we increase the index **i** by 1 and start again. If we never breakout until we reach **i > n-m**, then this means that the string P does not exist and we might return something like "not found"—in practice, we might return a negative value to represent the case where the string is not found.

All of this is a rather longwinded way of saying, "start from the beginning of string S and see if we can match to string P, if not, move one position forward and then try again." Trying to be precise can seem tedious until we get used to the programming language (or pseudo code). Hang on.

Like real world problems, computational problems have multiple possible solutions. For example, we could loop backward instead of forward. Sometimes, seemingly trivial changes in how we approach a problem can make a large difference, as we will see. Sometimes, seemingly trivial problems such as multiplying two numbers can require complex strategies to do it well (look up this problem in computer algorithm books). What is "doing it well" in computation? If we consider the code above, one simple trivial change is to consider stopping the inner loop as soon as we find a mismatch (i.e., breakout of the inner loop). It seems that would allow faster solution to the problem and therefore it would be solving it better. The foremost concern in solving a computing problem well is to minimize the use of resources. Two kinds of resources stand out in computing: ***memory*** and ***time***. For example, in the algorithm above it is clear that we need memory to store S and P strings and to be able to access it, essentially, randomly. By the last statement, I mean that when we ask for the value of S[*k*] some

mechanism exists to retrieve that information more or less instantly. We call such mechanism of storage, random access memory, or RAM. One can imagine that if S is very large, say the size of Genbank with some 400 billion bytes, such facile storage and retrieval might not be possible.

What about time? As you already know, time to compute something depends on a lot of different things such as the machine, how the machine does things, etc. Running the same computer game on a Mac, iPad, iPhone, all result in play at different speed. To analyze the amount of time required for a particular program can be a very tricky affair—and actually important. For example, recently I wanted to know how much time it would take to analyze some next-generation sequencing data (more on this later). This is a very large dataset, typically on the order of several hundred GB or so of input/output data. The program also runs on multiple processors. Because it reads and writes many files simultaneously from multiple processors it is fiendishly hard to know how much time the program will take. Actual time depends on the particular dataset, the memory state of various CPUs, the network interconnects (because the storage system is connected by a network), what disks were being accessed, and so on. I especially wanted to benchmark the time on cloud computing facilities. The virtual machines for cloud computing can have very variable actual hardware environments depending on when you access the machines, so it was doubly hard to figure out the computing time. In cases like this, the only feasible approach is an empirical approach; that is, try running the program a large number of times over a suitable number of test datasets and come up with a statistical characterization.

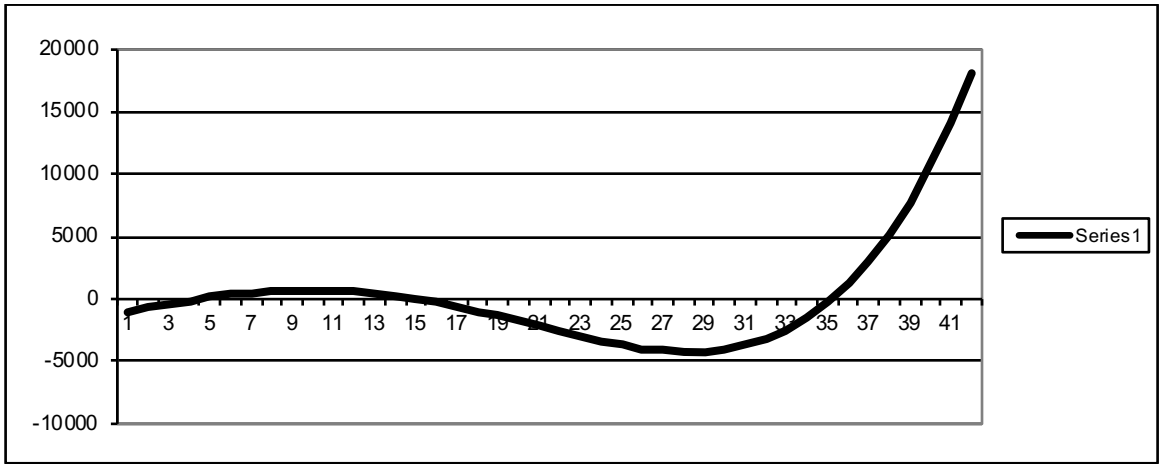
Any real-world program should be benchmarked by an empirical approach no matter what a theoretical analysis (to be learned next) tells you. However, there are many pitfalls to watch out for as well. For example, running the program may be costly—processing a single next generation sequencing dataset can take ~1,000 CPU hours. This means we cannot do a whole lot of empirical testing. A typical way around this is to process smaller problems (say smaller number of sequences) and extrapolate up to a realistic size. However, such extrapolation may breakdown in many ways. A machine may process 200 GB datasets with no problems and then suddenly take 10 times as long for a 500 GB dataset because of various storage bottlenecks. Processing a single data set might take 100 sec while processing 10 data sets may take an hour because of shared memory collisions, network delays, and so on. Careful empirical performance testing can be a whole enterprise and a research project. Regardless, all evaluations of an algorithmic approach should balance both empirical and theoretical analysis. This brings up the point for me the remind you that in any modeling or analysis we do, there is always a hierarchy of models/phenomena from low level to high level. For example, in a computing problem we have the low level to high level layers of micro-code driving the operation of a CPU, to assembly language, the higher programming language, code written in the programming language, to abstract description of the algorithm. In each case one might think that the lower level constrains what is possible to the higher level while higher level imposes a higher-order logic to the lower level. For example, the available micro-code and machine implementation constrains the efficiency of the code written in higher level language like Java while the code you write in Java imposes a logical flow to the execution of the micro-code.

Box: Asymptotic rate of function growth

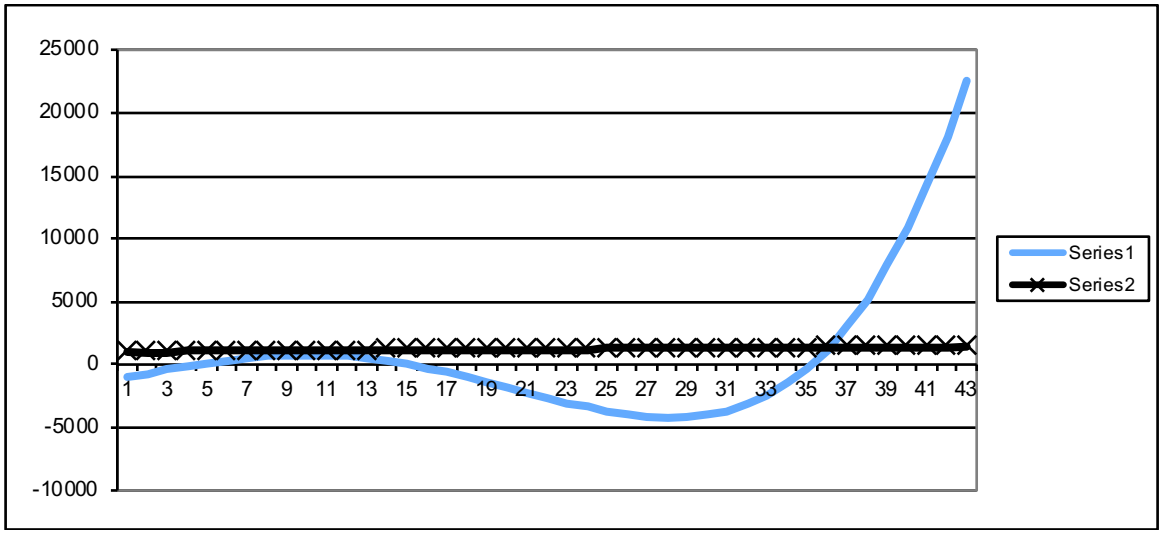
In discussing things like how much computing time is needed as the size of the input problem grows, we are typically not so concerned about details but kind of a rough estimate. To be more exact about "rough estimates", we introduce the notion of asymptotic bounds on some function. Suppose we have a complicated function say like this:

$$f(n) = -1000 + 300n + 2n^2 - 2n^3 + 0.05n^4$$

The curve for this graph up to $n = 42$ looks like this:

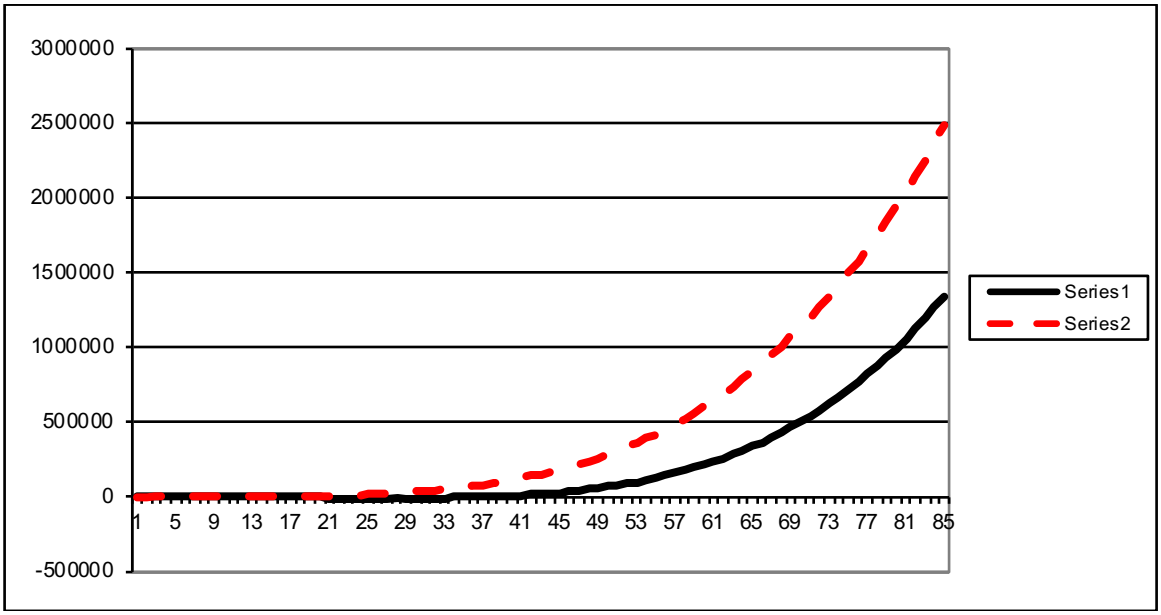


The main feature is that this graph goes up for a while and then goes down and then goes up again. Now let's compare it against a graph of $g(n) = 1000 + 10n$:



The *g(n)* graph is shown above marked by "x" symbols. What we notice is that *g(n)* is bigger than *f(n)* up to some point around 35 and then *f(n)* is substantially bigger. In fact, with a little bit of math we can show that after $n = 35$, *f(n)* is <u>*always bigger*</u>. Now suppose that *g(n)* and *f(n)* describes the time complexity of two algorithms for a *n*-sized problem. At small problem sizes, the algorithm with *g(n)* time complexity takes longer but once the problem size become bigger than a critical amount, the algorithm with *f(n)* time complexity takes much longer. Since we usually care about the general behavior of the system, it is useful to characterize the behavior of computation ignoring the irregular and potentially complicated behavior at the small sizes. So, this motivates us to only consider the behavior of the system when $n > crit$, where *crit* denotes some number.

Now consider the function $h(n) = 0.05n^4$. This function is similar to *f(n)* except for the lower order terms. The following shows the graph of *h(n)* (red) and *f(n)* (solid):



Now that we are plotting out to $n = 85$ the numbers are so big that we don't notice the dips and valleys in *f(n)* anymore. The two graphs look very similar except for a little displacement. In fact, with a little bit more math we can show that the values of the two functions never differ by more than a constant factor, *c*. And, the graph of $h(n) = 0.05n^4$ never differs from the graph of $p(n) = n^4$ by more than a constant factor. Since $p(n) = n^4$ looks very simple, it would be nice to say that *h(n)* is pretty much like *p(n)* and since *f(n)* is pretty much like *h(n)*, we would also like to say that *f(n)* is characterized by *p(n)*.

All of the above discussion motivates the following definition:

We say $f(n) = O(n^4)$ if once *n* is greater than some number *crit*, $f(n) \le cn^4$ (where *c* is some constant factor). That is, we can always say the *f(n)* grows at a rate that is *less* than a constant factor of n^4. Or, more informally, *f(n)* is <u>*bounded above*</u> by n^4. We can, of course, bound *f(n)* from below. For example, *f(n)* is bounded from below by n^3, by which we mean *f(n)* will grow <u>*at least as fast*</u> as a constant factor of n^3. The function *f(n)* is also bounded from below by n^4 as well as by n^3. (And, if you think about it, bounded from below by n^2 and n.) If *f(n)* is bounded above and bounded below by the same function, n^4 in this case, then we say *f(n)* has the same growth rate as n^4. Typically, we want to know the worst-case scenario so we are interested in the growth that is bounded from above. That is, we want to know that $f(n) = O(n^4)$; that way we know that the performance of something described by *f(n)* will never be worse than n^4. But, often times, these bounds are also exact—that is, indicates the same growth rate.

In all of our discussion above, I used a concrete example, but as you should be aware of now, the $O(\cdot)$ notation is used for any general function. Thus,

$$f(n) = O(q(n))$$

means the function *f(n)* is bounded above by the function *q(n)*. We use the notation $\Omega(f(n))$ for bounded below and $\Theta(f(n))$ for bounded above and bounded below by *f(n)*.

Going back to the pseudo-code in Fig 1, we can now carry out a theoretical analysis of the computing time required to run this program. There are two **while { }** loops in the code, one nested inside the other. Therefore, it is clear that the statements inside the second **while { }** loop will be repeated the most number of times. We might reasonably think that the total time for the program will be generally determined by how many times those inner loop statements are carried out. It is easy to see that the two statements, **if(S[i+k] <> P[k]) then result = 0; increment k,** will be repeated *(n-m+1)m* times. Or, if *n* is much bigger than *m*, then *n-m+1* ~ *n* so the inner loop statements will be repeated about *nm* times. The whole reason to write a program is to solve the same problem multiple times on different inputs. The main important characteristic of the input for the pseudo-code in Fig 1, is the length of the two strings, *n* and *m*. We usually want to know how much time a given algorithm will take as a function of the characteristics of the input. Here, our analysis tells us that we expect the computing time to be (most likely) a function of *n* times *m*. In actual program execution, there are other statements besides the ones in the inner loop that need to be processed. For example, locating the data (the two strings), reading the strings into random access memory, etc. However, as *n* and *m* become larger and larger, these other steps will contribute less and less to computing time (with some possible exceptions as we discuss below). How an algorithm's resource use scales with the size of the input problem is called the ***computational complexity*** of the algorithm. (Or, the computational complexity of the problem; we will discuss the difference between a "problem" and an "algorithm" later.) Typically, we are interested in the asymptotic complexity—that is, what happens to time use or memory use when the input size becomes pretty large. The pseudo-code in Fig 1 has time complexity *O(nm)* and memory complexity *O(n+m)*, where the notation *O(.)* means "asymptotic upper bound" (see boxed text above).

We are still not quite done thinking about Naïve Search pseudo-code. We assumed that when we ask for "ith symbol of string S" by **S[i]**, we will have instantaneous access to that information (RAM). What if such memory cannot be used? Suppose all the information is recorded on a physical tape and each time we need the *i*th information, we need to physically move the tape to the *i*th position from the beginning? Then the time complexity bound of *O(nm)* doesn't tell us the right picture. What about the index variable **i**? Will our computing machines behave the same if **i = 0** versus **i = 300 billion**? Will they properly handle index values like *2 trillion*, a number that might occur in genomics data? Most modern computers provide complicated architecture to allow access to memory in an arbitrary (i.e., random) manner and also to provide variables that can count large numbers. But, there is a limit, sometimes a fairly important limit. Machine implementation of computing involve finite modules that carryout finite operations. Things are fine while the problem size is within these finite limits but if the problem size gets bigger, much more machine operations have to be carried out—sometimes very much more. Available memory is often structured such that a certain amount of memory can be accessed very fast, others not so fast (e.g., placed on hard disks, whose recall/access can be very slow). In addition, how a given programming language handles certain kinds of data structure may be very different. Previously, we postulated that the main determinant of the Naïve Search algorithm speed might be how many times the inner loop statements are executed. Suppose S is a very long string, say *n* = 500 billion. If we naively try to read this into a single large string variable, simply reading the data may take much longer than computing the statements in the inner loop. Scripting programming languages, such as Perl, R, and Python, tend to have difficulty with very large memory use. One thing we can see from the Naïve Search pseudo-code is that for at least one of the strings, we don't really need to access the memory in a random manner. That is, string S is examined from left to right (index value *0* to *n-m*) without ever going back; and, at the time of use, we only need to access *m* number of positions (the length of P). Therefore, we should be able to implement this algorithm with memory complexity *O(min(n,m))*; that is, we only need RAM in proportion to the shorter of the two strings. Small memory usage may allow some algorithms to be much faster than other algorithms even if theoretically they have identical asymptotic time complexity.

With all that we are now ready to see if we can improve upon Naïve Search algorithm. The main thing we want to do is to see if we can reduce the number of comparisons. The naïve part of Naïve Search algorithm is in comparing against string P and then moving one step along string S and doing the same thing all over again. It seems we should be able to avoid all this. We already considered breaking out of the loop as soon as we find a

mismatch. We now use a seemingly trivial but a very important change, which turns out to have a very big effect. Inside the inner loop of the Naïve Search algorithm, we scan the string P left to right against left to right positions of string S, starting from the offset **i**. What if we start the scan from right to left? Suppose at the very first right most symbol of P, we found it did not match the corresponding position in S, call this position S[*k*]. We can immediately move the string P rightward (i.e., increase the offset) all the way to a position within P that we do know matches to the symbol in S[k]. Here is an example:

```
    123456789
S: "AAACTTGTT"
P:    "TAC"
```

String P is currently at offset 3. Scanning from the right we see the symbol "C" in string P does not match the symbol "T" at S[6]. This tells us we can immediately advance string P like this:

```
    123456789
S: "AAACTTGTT"
P:      "TAC"
```

Now, string P has moved to offset 5 from offset 3, jumping over evaluating offset 4. (Why are we able to do this?) Thus, we've saved a certain amount of redundant comparisons. Why not jump to offset 6; i.e., a full frame of the string P? Since the "T" in P[1] does match S[6], the position we evaluated, we have to make sure no extension from that position results in the full match of the string P.

This seemingly simple but great idea is small part of the core idea in the famous Boyer-Moore algorithm, developed by Bob Boyer and J. Strother Moore in 1977. As we noted before, a mismatch in the string P tells us to stop processing any more of the string. The key idea in Boyer-Moore algorithm is that this also tells us something about the string S and how to skip some of the processing of string S. Going from right to left, especially gives us the information about the maximum amount of skipping we can do in string S. We will do the full Boyer-Moore algorithm later but for now, here is the pseudo-code for a simpler version:

```
i = 0
while (i <= n-m) {
    line A: k = m
        result = 1
        while (k > 0) {
                if(S[i+k] <> P[k]) then {
                        result = 0
                        increment i by bad_character_skip(S[i+k], k)
                        go to line A:
                }
                decrement k
        }
        if result = 1 then return i
        increment i
}
return "not found"
```

Fig 2

Many things have happened in the pseudo-code for Fig 2 compared to Fig 1. First, as can be seen the inner loop starts from the end of string P($k = m$) and moves left (decreases the index k). Second, inside the inner loop is the statement

increment i by bad_character_skip(S[i+k], k)

Here, there is a function called **bad_character_skip()** that has not been defined yet. It takes in two values, the symbol at S[i+k]—this is the position that has been found to not match the current P[k] symbol, and the value of k—the current position within the string P. This function will tell us how much to change the offset so that the string P will be shifted to the right, either by the full string, or until a position within P that matches the symbol S[i+k]. If this seems confusing look back at the example above. We assume this function will be pre-computed. The pre-computation will require a table of size p×m, where p is the number of symbols in the alphabet. I note here that if the number of symbols is large (like the English alphabet and all the symbols one might find in a MS Word document) and the search string is relatively long, then this table can be pretty unwieldy. (The original version of Boyer-Moore algorithm uses a much smaller table, discussed below.) Since the number of symbols in DNA or amino-acids is manageable, we can easily consider a p×m table. Here is an example for the search string "AAACGT", where the rows index the mismatched character at S[i+k] and the columns index k, the position within P where the right most mismatch occurred:

S[i+k] \ k	1	2	3	4	5	6
A	0	0	0	1	2	3
C	1	2	3	0	1	2
G	1	2	3	4	0	1
T	1	2	3	4	5	0

You should convince yourself that this table can be computed only from the string P and does not require information from string S. Pre-computing this table means that we will assume when we need the value of **bad_character_skip()**, we will get it instantly without additional computing. Of course, we need to store these pre-computed values, so we will need to use p×m RAM memory to achieve this instant access.

In design of algorithms there are many ways to tradeoff memory use for computational time. In learning to multiply multi-digit numbers in grade school, we were made to memorize the single digit multiplication table. Then we were taught an algorithm to apply the memorized multiplication table (what we call a "lookup" table) position by position. If we had memorized a larger table, say all two digit multiplications, we would be able to do longer multiplications much faster. If we didn't memorize our multiplication tables, we would have to do a lot of sums for even single digit multiplications and it would take forever. In fact, the architecture of a computer's CPU employs a lot of such lookup tables to judiciously speed up certain operations while trying not to blow up the complexity of the CPU.

The third new thing in the pseudo-code in Fig 2 is the statement "**go to line A:**". It means to move the execution of the program to the line that is labeled "**line A: k = m**". That is, break out of the inner loop and start over again at "**line A**" with the new value of the index **i**. (Yes, sometimes when we are working with a limited programming language, a "**go to**" statement can be useful.)

Okay, what do we expect from this new algorithm? A moment of thought will show that it depends on the contents of the string S and P. Suppose S = "`AAAAAA…. A`"; say, a 100 A's and P = "`TTTT`". As soon as we compare the right most `T`, we will shift the entire string P by 4 positions. So, we will end up doing ~100/4 symbol comparisons. If P = "`TTTTTTTT`", we will shift the string P by 8 positions and end up doing ~100/8 comparisons, etc. This is great! In this best case scenario the time complexity of this algorithm is O(n/m). This is a key feature of Boyer-Moore algorithm: the search can be faster the longer the search string! Now suppose P = "`TAAA`". Then we will not know that P does not match the first 4 symbols of S, until we get to the very last (left most) symbol of P. Then we shift by one. So, in this worst case scenario we are right back to doing O(nm) comparisons. What do we expect if both strings S and P are uniformly random strings of `A`, `C`, `G`, and `T`? I leave that for you to solve (hint, try with the simplest case first where we assume the right most symbol in P immediately mismatches; how many positions will we skip on average?)

Thinking about this problem further, it seems like we should be able to do better than O(*nm*), even in the worst case. If P = "TAAA" here is what we would have compared at offset = 0:

```
   123456
S: AAAAA..A
P: TAAA
```

Starting from P[4], we have to count down to P[1] to know that we have a mismatch. The algorithm above, then shifts S by +1 and starts at P[4] aligned to S[5] and counts down again. However, when we are doing this we are throwing away information we already know. That is, we already know that S[2-4] matched P[2-4], and none of these were the symbol "T". So, our intuition is that we should already have the information to skip by +4 positions. In fact, this is true.

The full Boyer-Moore algorithm tries to match the search pattern, P, from some offset position in the database S, moving from right to left. When a mismatch has been found, we know we can increase the offset, effectively moving the pattern P in the right hand direction against S (or, equivalently moving S left against P). Suppose the length of P is *m* and the mismatch happens *k* positions left of P[*m*]; i.e., at P[*m-k*]. Since we know all the symbols of P, we know the corresponding letters of S from P[*m-k+1*] to P[*m*]. Thus, we know how to best shift P. The idea of a **suffix match table** is to use this information.

To make things concrete, let's start with an example:

```
  123456789
S:TATTCGGTT
P:GCGACG
```

The mismatch has happened in the P[4]-S[4] position. But, we already know that P[5,6] = "CG" has matched and therefore the corresponding substring in S is "^ACG" where I used the symbol "^A" to denote "not A". (Actually, when we are actively scanning we know that the precise corresponding substring in S is "TCG"; we will see below how that information can be used to skip even further.) Looking at the internal structure of P by eye, we know there is another "CG" left of the first "CG"; therefore, we should be able to increase the offset until that "CG" aligns with the "CG" we already know is part of S:

```
  123456789
S:TATTCGGTT
P:   GCGACG
```

At this point, we start at S[9] and P[6] again to try matching. The key is that we can compute these offsets by an examination of the internal structure of P. Basically, we are looking for the repeat structure of P. We want to enumerate all the possible repeat structures. I will first show a suffix match table and then we can discuss how this is constructed. We enumerate the mismatch cases from right to left of the string P, and then what should happen when we find a mismatch:

Position of first mismatch of P, scanning from right to left	Implied pattern in S	Action to be taken
P[6]	^G	increase offset by 1 (i.e., move P right by 1)
P[5]	^CG	increase offset by 5
P[4]	^ACG	increase offset by 3

P[3]	^GACG	increase offset by 5
P[2]	^CGACG	increase offset by 5
P[1]	^GCGACG	increase offset by 6

How do we know what action to take? The basic idea is to find out if there is a pattern starting left of the current mismatch position that matches the pattern in S (the first column of above table). For example, we might have:

```
S:...TCG
P:GCGACG
```

To the left of "ACG" in P, we have "GCG", which matches "^ACG" and is at 3 positions left of the current position (thus, increase offset to 3).

Wait, why are we not looking for "TCG", which we know is S[5-6]? Because, we want to compute these offsets by only looking at P without consideration of the mismatch symbol in S. We will see later, that we can use the information that the mismatch symbol was T and not {C, G, T} in a different manner.

If there is no match left of the current evaluation point, there still could be some prefix left of the current point matches the suffix right of the current point. That is, we can slide the string P right up to the point until it overlaps with known parts of S (represented by suffixes right of mismatch point).

A better way to summarize the two cases we just discussed is to consider the pattern at the point of mismatch. We put down the original P string (not S) and then slide the mismatch pattern to the left, until we get a match. When we are doing that, we assume that the original string is padded to the left by "wild card" symbols (i.e., symbols that match any symbol). Like this:

```
P:.....GCGACG
0:         ^ACG
1:        ^ACG
2:       ^ACG
3:      ^ACG
```

This tells us that there is an internal repeat of ^ACG pattern +3 offset from the current mismatch. We try this with the case that we find a mismatch at P[2]; i.e., ^CGACG

```
P:.....GCGACG
0:       ^CGACG
1:      ^CGACG
2:     ^CGACG
3:    ^CGACG
4:   ^CGACG
5: ^CGACG
```

So the calculated offset is +5.

To fix ideas, we do one more example with P = "TAAAA".

Position of first mismatch of P, scanning from right to left	Implied pattern in S	Action to be taken
P[5]	^A	increase offset by 4
P[4]	^AA	increase offset by 3
P[3]	^AAA	increase offset by 2
P[2]	^AAAA	increase offset by 1

P[1]	^TAAAA	increase offset by 5

The suffix match table has an entry for each of the possible cases where the mismatch was found in *m, m-1, m-2,…,1* positions of the string P, where *m* is the length of P. The way I've described the construction of the suffix match table, you would have to do a search through the string P for each of those mismatch cases and the algorithm might take something like $O(m^2)$ time to construct this table. Can you think of how to do this in less time?

Okay, now one more part. In the modified Boyer-Moore that I gave in Fig 2, we computed a fairly large bad character table. Here, we compute a simplified table, only with respect to the last (right most) symbol in P. Basically, we want to compute for each possible symbol the right most place it can be found in P. For example, with P = "GCGACG", then we have: A = 4, C = 5, G = 6, T = 0. Suppose S[i] was a mismatch to P[j] at some point in the algorithm, then we are supposed to look up S[i] in the bad character table. Then we compute

max(1, j - bad_char_tab(S[i]),

where bad_char_tab(S[i]) means look up the value in the table. Note that depending on where you are j – bad_char_shift(S[j]) can be a negative number. If it is, we set the offset shift to 1.

We will call the two strategies the "bad character rule" and the "good suffix rule". Finally, the Boyer-Moore algorithm in words:

Match P to S, right to left, from some offset position in S. If a mismatch is found, compute *shift* = max(bad character rule, good suffix rule). Increase offset by *shift* and start matching P to S at the new offset position, right to left.

The Boyer-Moore algorithm is the gold standard in exact string searches. There are other cases, which we will discuss in a moment, where we might want to use a different strategy but for straightforward single string search, this would be the algorithm of choice. If we go back to the examples we discussed before, say looking for "GTTTT" in a string of 100 A's, you can see that the bad character rule will be evoked immediately and we will evaluate at 20 positions and we will find that the string does not exist. If we were looking for "TAAAA" in a string of 100 A's, the suffix match rule will be evoked and again and this time we will evaluate no more than 100 positions and conclude the string does not exist. However, it turns out there can be some pathological cases. Suppose we were looking for "AAAAA" in a string of 100 A's and want to find all of them. We find a match at the first position, but none of the rules allow us to move more than 1 position. So we end up doing $O(mn)$ comparisons again. There are additional fixes to this problem. It turns out that with all the trimmings, the Boyer-Moore type of strategy will have the worst case time complexity of $O(n)$.

The pathological example just discussed points to a very important idea when coming up with solutions to problems—whether the problem is algorithmic, statistical, or mathematical in nature. We have to always try hard to consider the extremes of the input cases: the best case, the worst case, etc. What we want is a solution that behaves nicely in some sense over all possible cases. For example, we might want the solution to behave nicely even in the worst possible case, or, behave nicely on the average. Or, to behave in a smooth predictable way from the best to the worst so that we know when it is starting to behave badly.

A final comment on the Boyer-Moore family of algorithms. As mentioned, the algorithm has a very nice property that it tends to become more efficient (on average) when the length of the string pattern sought is longer—this is a very desirable thing indeed. However, the gain in efficiency tends to be lower when the size of the alphabet is small. As you can surmise from the algorithm, when we have a mismatch, it is best when there is no pattern or symbol in rest of the string P. Larger the alphabet size, more likely this will be. Small alphabets like DNA tend to have a lot of possible matches left of the mismatch position, then the size of the offset shift tends to be small, losing efficiency. Thus, our work is not done.

Unit 3: String search by structured data

We now consider the case where we have some common fixed database of strings like Genbank and we would like to search the database over and over again. In these cases, we might try to organize our database in some way such that multiple queries can be efficient. Organizing the database might take a lot of effort, but if we are going to carry out millions of searches on it, it might be worth the effort. The canonical example of such organization is your standard dictionary or encyclopedia. Effort has been put into the dictionary/encyclopedia to organize the entries in an alphabetical order so future searches can be efficient. In fact, the usual dictionary/encyclopedia also has "pointers" to sections so you can cross-reference from one section to another. Organizing information into particular structures that facilitate access is called (making) "Data Structures". There is a large volume of information on data structures in computer science and the data structures are intimately tied to algorithms. I should also note that many data structures can be theoretically equivalent but can have a large impact on performance in actual programming code, especially as a function of the programming language.

Let's start with some simple examples of data structures. You can study the subject in detail in many computer science courses. We start with unstructured data: {0, 2, 4, 3}. This is a set of four numbers and you will remember from math that a set only "contains" the objects and there is no meaning to the order. If I wanted to know whether the number 4 is a member of the set, I would have to search through the set—somehow. An "**array**" is the most important kind of data structure. It is equivalent to taking the unstructured set and giving it an order (making it a k-tuple): (0, 2, 4, 3). (Recall in mathematical notation a set is denoted with {} brackets while an ordered set is denoted with ( ) brackets.) Now, the 3rd item in the 4-tuple is the information "4." The tuple can be given a name (i.e., a variable) so we might write N = (0, 2, 4, 3). Then N[1] = 0, N[2] = 2, etc. Programming languages implement this in various notations (syntax). In Perl we would write @N = (0, 2, 4, 3) and then retrieve the information using the syntax $N[1] = 2. (Unfortunately, Perl starts array indices from 0 so this example is shifted from my preferred convention). The key idea of an array is that the locations of the objects are indexed by natural numbers so we can use arithmetic operations to retrieve information. For example, we can use a variable for natural numbers, e.g., $i (continuing with Perl syntax, where the variable names are preceded with the symbol "$"). If we set $i = 0, and then retrieve $N[$i], we get the value 0 from the array @N. (In Perl, array variables are preceded by the symbol "@" when referring to the whole array and preceded by the symbol "$" when picking out individual members of the array.) Now we can carry out arithmetic operations on $i, say, add one such that $i = 1 then $N[$i] = 2. If you have never programmed before, this is VERY IMPORTANT. The same symbolic notation "$N[$i]" retrieves different pieces of information dependent on value of $i. If we weren't able to use such higher-order notation (and data structures), programs would be extremely unwieldy.

An array assumes that there is a machine-implemented method to access the ith information. Imagine that your information is written on a stack of index card and I wanted the 98th card out of 500. You can see this might not be so easy without some special provision. A simpler kind of data structure is where the information has order but it can only be accessed from the "top" or the "bottom" or both. A "**stack**" is a data structure that is just like a stack of index cards. You can store information by putting a new card on the top and retrieve information by taking off and reading the top card. Thus, a stack has the characteristic of LAST-IN-FIRST-OUT (LIFO); that is, the last piece of information in the stack is the first piece of information that can be retrieved. When you retrieve the information, it is also lost from the stack. Putting information into the stack is called "push" and retrieving information is called "pop" (e.g., popping from stack X). A "**queue**" has a little bit more so that you can access the last item in the order. Information is again pushed in (or sometimes called "enqueue") and when the information is popped (also called "dequeue"), the first piece of information that was stored is retrieved. That is, we get FIRST-IN-FIRST-OUT (FIFO). It may seem like these are very primitive structures (and they are), but both the stack and queue are used often in programming (and in the CPU machine operations), typically for storing a list of things to do.

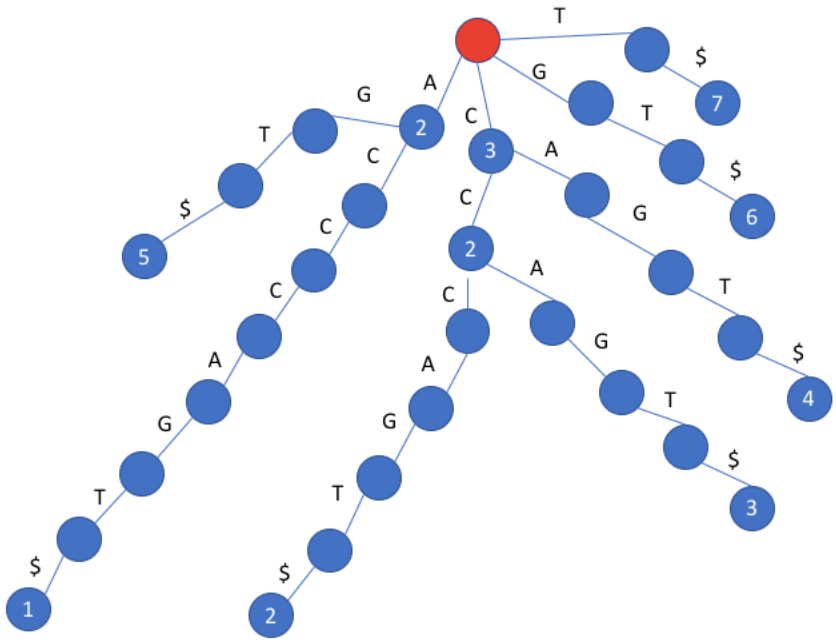
As mentioned, there are many different kinds of data structures used in algorithms and we will continue with two additional examples that are commonly used in computational biology. While an array allows us to retrieve information using natural numbers and arithmetic operations, many kinds of information are not naturally organized into numerical order. However, we often have information that come in pairs, say FIRST NAME + LAST NAME; or, say GENE NAME + DNA SEQUENCE. We can represent this as a set of 2-tuples: for example, {(A, 0), (C, 2), (G, 4), (T, 3)}. An associative array gives us a way to store such paired information. We assume that each 2-tuple is the pair (Key, Value) so that if we denote %N = (A, 0, C, 2, G, 4, T, 3) (here, Perl denotes associative array with variable names that is preceded by the symbol "%" and each pair of objects in the list are assumed to be Key, Value pairs), then $N{'C'} retrieves the information value "2". (Perl uses the syntax $N{ } to access elements of an associative array.) Associative arrays are often used to store information that is commonly referenced by strings of letters rather than natural numbers. Examples include (Gene Name, DNA string), (DNA string, Gene expression), (DNA string, occurrence frequency), etc. At first glance, this seems like the way to go. Just put all of our information in (Key, Value) format—but, we will see later that this requires a lot of additional computing and difficulties.

Arrays, stacks, and queues store information in a one-dimensional linear order but efficient retrieval of information is often aided by multi-dimensional structure or a more general structure. In particular, storing information in a tree graph structure is very useful in a wide variety of algorithms. We first briefly discuss graphs and trees as a discrete object. A graph consists of two sets, a set of **vertices**, V, and a set of **edges**, E, such that we write G = {V, E}. Edges connect the vertices to each other. So, if the set of vertices are V = $\{v_1, v_2, \ldots v_n\}$ then we can depict an edge by the pair $e = \{v_i, v_j\}$; that is, the edge e connects the two vertices v_i and v_j. An edge may be directed in the sense that the edge $e = (v_i, v_j)$ now depicts connections from v_i to v_j but not the reverse connection (v_j to v_i). If a graph consists of directed edges than we call it a **di-graph**. Each edge connects only two vertices but each vertex may have many edges. The number of edges of a vertex is called its **degree**. If it is a di-graph, then a vertex will have in-degree and out-degree. A special type of graph is a tree graph. In a tree graph the structure of the connections are such that there are no circuits between vertices. That is, you cannot use the edge connections to go from one vertex to another vertex and then return to the original vertex using a different path. Therefore, in a tree graph, every pair of vertices has a unique path connecting them. A tree graph may have a root, which is a special vertex and a path in the tree can be considered to be moving away from or towards the root. A tree graph will have one or more leaf-vertices, which are the vertices at the terminal end of a path leading away from the root vertex. The vertices and edges of a graph may have associated information and/or labels. A formal way of talking about such information is consider a map from the set of vertices (or edges) to another set that contains the information. So if we have the vertex set V = $\{v_1, v_2, \ldots v_n\}$ and the set of real numbers, $\mathbb{R}$, then a map $\rho: V \rightarrow \mathbb{R}$ might be a real-valued annotation of the vertices. For example, the vertices might represent genes, then $\rho(v_1) = 1.67$ would represent the idea that vertex v_1 has the expression value 1.67. The range of the map for the vertices (or the edges) might be DNA strings, names (e.g., species names as we will see later when we discuss phylogenetic trees), numbers, pictures, etc.

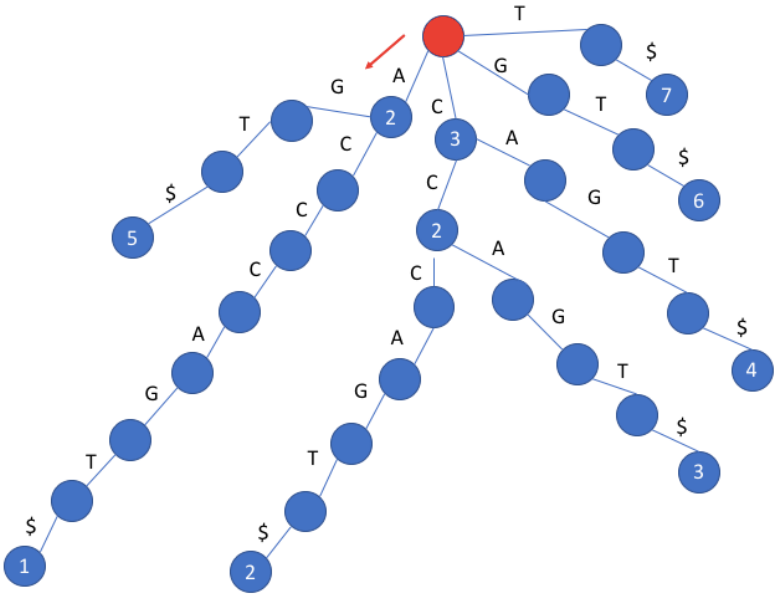
An important data structure for accessing strings is called a suffix tree, or a prefix tree depending on how we arrange things. A prefix tree is just the string inverse of the suffix tree and we can use them equivalently; thus, for this text I will just deal with suffix trees because it relates better to a topic we will discuss later (suffix arrays). (More recently, these structures have also been called a "trie" instead of a tree, from "re**trie**val" to give it a technical flavor.) Suppose we have the string **S** = "ACCCAGT". We start with enumerating all of its suffixes.

```
ACCCAGT$
 CCCAGT$
  CCAGT$
   CAGT$
    AGT$
     GT$
      T$
```

I put the symbol "$" at the end of each suffix to denote the end position. The following is the suffix tree:



The picture shows a graph with vertices and edges. Each edge is labeled with a symbol. The red circle is the root of the tree. The vertices (also sometimes called nodes) are decorated with other information, which we will discuss in a minute.

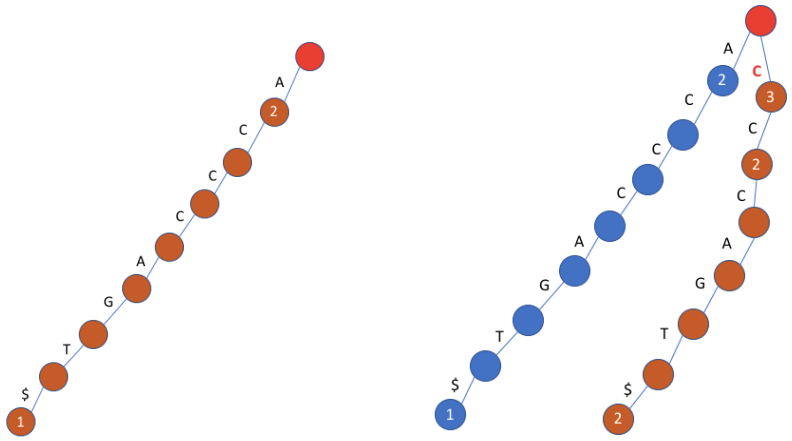


Consider what happens if you walk from the root of the tree toward the leaf of the tree (the last vertex in the path from the root). We walk from the root on the path marked by the red arrow in the figure above.

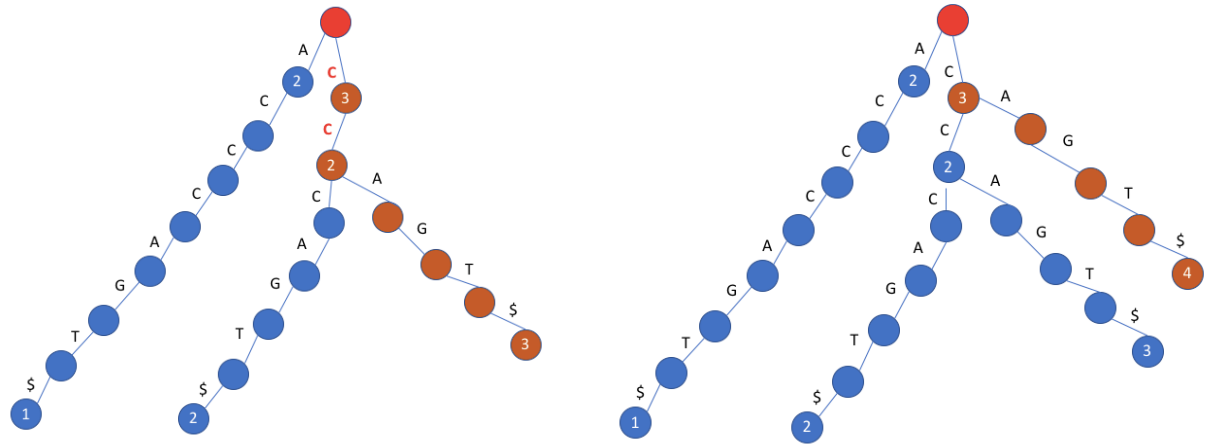
We first encounter an "A" symbol along the edge. If we now go down the left child branch of the vertex below this edge, we encounter "G", then "T", then "$". This translates into the suffix "AGT$". If we go down the right child vertex, we will see a sequence of symbols "CCCAGT$". Concatenate this with the first "A" and we get

"ACCCAGT$", which is the whole string and also a suffix of **S**. If you continue this kind of path walking from the root to all of the leaves of the tree you will see that we will recover all of the suffixes of **S**. I have annotated the leaves of the tree with the index number of the suffix. So the "AGT$" path ends at a leaf marked "5", "ACCCAGT$" path ends at a leaf marked 1, and so on.

Before we discuss what we can do with the tree, I will show you how to construct one by hand: Start with the longest suffix and then construct a set of vertices from the root, as many vertices as the length of the suffix. Put the symbols of the suffix on the edges with the last edge marked with the symbol "$". We then take the second longest suffix. We examine the symbols on the existing tree to see if this suffix can be "hung" off an existing edge on the tree we have made so far. If this is not possible we start a new branch from the root, like this:

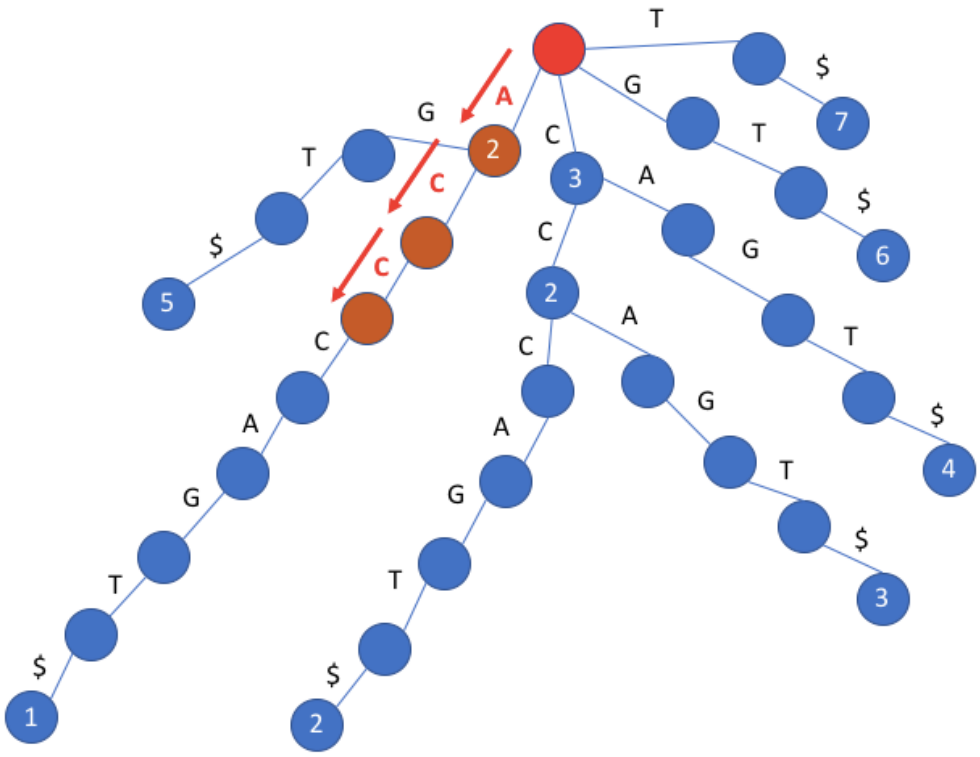


We continue adding branches of the tree that correspond to the next suffix of the original string. At some point in the tree construction, some suffix will naturally extend from an existing branch:

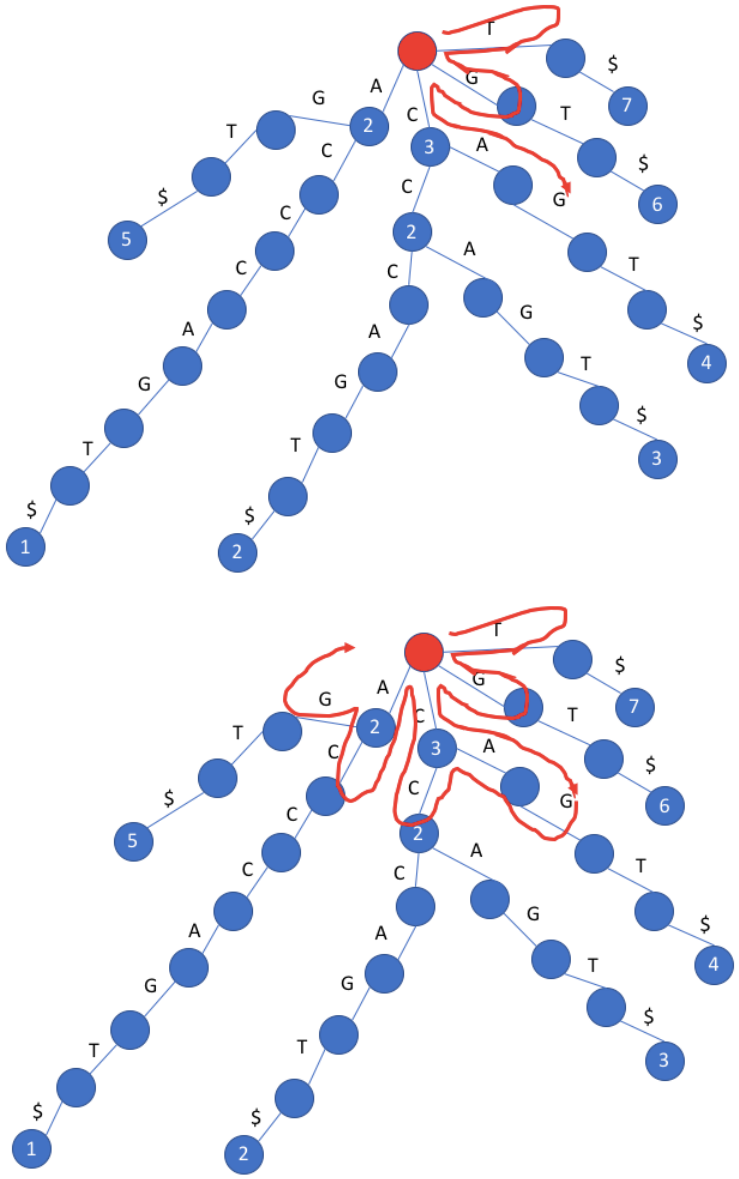


In the picture above the suffix "CCAGT$" can be added to the tree using the existing "CCCCAGT$" path as shown above. The same for "CAGT$". We continue until we finished inserting all of the suffixes.

What can we now do with this tree? Suppose we were looking for the string "ACC" within the string "ACCCAGT". We start at the root of the tree representing "ACCCAGT" and find the edge with "A", then keep going down the tree looking for "C" to make "AC". Keep going until we find "ACC". It took only the same number of operation as the length of the string that we were searching for. Notice that the leaves of this tree has numbering of the suffixes that one gets moving from the root to the leaf. So, from that information we learn that the string "ACC" starts at the 1st position because the branch where we found "ACC" was the branch leading to the 1st suffix. Suppose we were looking for just "C". Again, we start at the root and look for an edge with "C". The edge with "C" has three children that end at leaves marked, 2, 3, and 4. This tells us that the symbol "C" can be found in starting positions 2, 3, and 4. The tree immediately reveals all the repeat structures of the original string. For example, looking at this tree tells us that there is one "T", one "G", three "C"s, and two "A"s. It also tells us there are two "CC"s, and so on.



What else can we do with the tree? We can use the tree to implement inexact searches. Suppose we want to search for "TAG" allowing for up to one mismatch? Then we go down the tree from the root looking for "TAG". When we find one mismatch we keep going down the tree until we either complete the search or find a second mismatch. If we do find a second mismatch we backtrack up the tree and go down another branch, until we find the string that satisfies our one-mismatch condition or we fail. The red line in the figure below shows the search path for "TAG" where we continue the search even if we find a mismatch until we find "CAG". We note that we can't just stop after we find "CAG". We have to continue the search since there could be "GAG" and "AAG". The next figure shows completing the search until we are sure there are no other substrings that match our criterion. For the naïve search algorithm we discussed at the beginning we can also implement a *k*-mismatch algorithm by simply counting the number of mismatches as we do the exhaustive search. But the tree approach here will be much more efficient (how much?).



As we have just seen, the tree is an extremely useful structure. What are the downsides? Well, the first downside is that the way I've described the construction, it takes $O(n^2)$ time to construct the tree, where n is the length of the database string—this is the amount of time required to enumerate each suffix (or preffix). Fortunately, there is an algorithm to construct this tree in $O(n)$ time, which is too complicated for this course. There is a more serious problem in that storing such a tree requires more memory than the original length of the string. The tree shown above is much bigger than the original string (what is the space complexity?). This can be a problem when the database is very large. It turns out that the structure of the tree can be encoded by using various pointers such that the space usage is $O(n)$, although it can be a fairly large multiple of n. There is also a more compact data structure that can store the tree information called a suffix array (or a prefix array). We will see the suffix array structure later when we treat the Burrows-Wheeler Transform for next-generation sequence data. Unfortunately, suffix arrays are not as simple to access as walking down a tree graph. Regardless, then given something as nifty as the tree data structure, why would we even consider something like Boyer-Moore? First, it takes $O(n)$ time to construct the tree, which isn't an improvement on the Boyer-Moore. And, the Boyer-Moore algorithm uses less extra space (aside from the database string itself), and it is also very simple algorithm. It is only when you use the tree multiple times that it proves itself useful since each search takes time proportional to the search string, not the database. Second, remember that in the best case Boyer-Moore has the time

complexity O(*n*/*m*), which can be very good—faster than the size of the database (we call this a sublinear algorithm).

Unit 4: Finite State Automata and computing

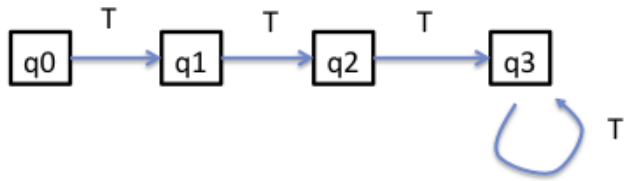
We now consider a very different approach to the string search problem. The motivation here is that a particular approach that we call Finite State Automata (FSA) turns out to be a component of a fundamental model of computing, leading to among other things, the very famous Turing Machine (TM). Furthermore, the FSA approach gives us a standard way of searching for (and defining) a family of abstract string patterns called the "regular grammar" family.

Automata are "machines" (in a conceptual sense) that have "internal states" and read inputs. Upon reading an input symbol, they do something based on the combination of the input symbol and the current internal state and the result is, voila, computation. A FSA, one of the simplest type of automata, is defined by: (1) a set of finite internal states, (2) a set of finite input symbols (or letters), and (3) a set of transition rules that reads "if machine is in state *p* and reads symbol *v* change to state *q*." (A Turing Machine has just a few more rules.) Useful computation is obtained by declaring an "accepting state", which is supposed to represent the completion of some specified task. (We may have multiple accepting states in some FSA.) The main characteristic of an FSA is that it has no memory except the current state. Future behavior depends only on the current state and no past states or past events. You will see later that there are other memory-less models of processes in other disciplines (e.g., Markov models in probability), which share structural similarities to FSA.

The FSA is often specified by giving a table of transitions where the columns represent possible input symbols, the rows represent the possible machine states, and each cell is the next transition state given the combination of row machine state and column input symbol. For example, here is a FSA over {`A, C, G, T`} that processes a DNA string and reaches the accepting state if it sees "`TTT`". (Note that this machine continues to process the string after the first occurrence of "`TTT`".) This FSA has four states labeled **q0-q3**. The **q0** is the initial starting state of the machine and **q3** is the accepting state.

	A	C	G	T
q0	**q0**	**q0**	**q0**	**q1**
q1	**q0**	**q0**	**q0**	**q2**
q2	**q0**	**q0**	**q0**	**q3**
q3 (accepting)	**q0**	**q0**	**q0**	**q3**

This transition table can be depicted (see next figure) with a graph where the vertices represent the machine states and the directed edges represent the transition rule conditioned the input symbol (indicated by labels on the edges). Note that since there are four possible input symbols for the DNA alphabet, every vertex should have four edges coming out of the vertex. However, since depicting this can be messy, by convention, we agree that if a particular possible input symbol does not have a corresponding edge drawn, it means to return to the initial state. That is, in the figure below, **q1** has only one edge labeled with the input symbol "`T`". If the input symbol is "`A`", "`C`", or "`G`", then the transition is back to **q0**.



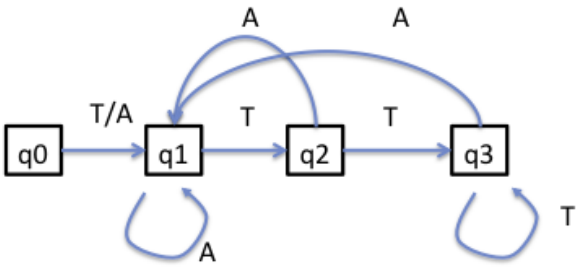
Let's try running the FSA above when the input string is "ATTCTTTG". When following the rows of the table below, read it as "*the FSA starts in* [State of FSA] *and reads the* [Symbol processed] which induces a transition to [State of FSA] *in the next line.*"

Step	State of FSA	Symbol processed
0	**q0**	A
1	**q0**	T
2	**q1**	T
3	**q2**	C
4	**q0**	T
5	**q1**	T
6	**q2**	T
7	**q3**	G
8	**q0**	END

You see that at step 7, the machine reaches the accepting state **q3**. Imagine that a bell rings or something at this point; then you know that a substring "TTT" has been found within the input string. After that point, the machine reads a new symbol "G" and goes back to the initial state **q0**. If you look at the figure, you see that from **q3**, if the input symbol is another "T", then the machine moves to **q3** again. This is for cases where we might have "…TTTT…", in which case we have two instances of "TTT": "…TTTT…" and "…TTTT…"

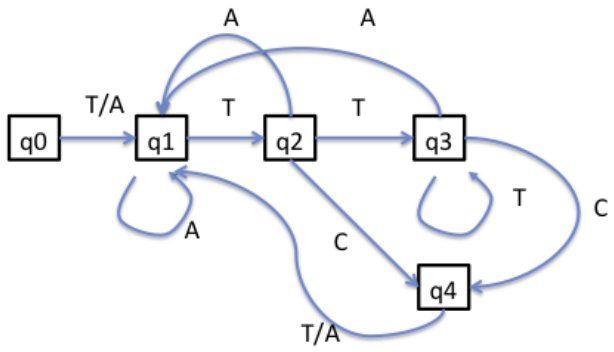
How do we make the transition table? How did I know how to fill in the transition table (and therefore, the transition graph) to build a FSA that finds "TTT"? One way to think about this is that the class of all possible FSA is a programmable computer, albeit a simple one. A particular instance of FSA is a program in the programming language of FSA. Making a transition table can be considered the same as "programming." For simple kind of FSAs that search for fixed strings, we can describe a step-by-step formula for constructing FSAs that processes the strings. That is, we can construct a program that generates FSA transition tables—you can think of such a program as a **compiler** that translates string patterns to FSA transition tables. A compiler that generates FSA transition tables for a fixed string is relatively easy to write. Compilers for much broader class of string patterns, like the examples shown below, can also be written. Here, we will not study how to write such compilers but concentrate on understanding how to come up with the transition graph for any string searches. It is much easier to draw such figures (graphical programming, so to speak), than to write compilers for FSA.

The great strength of FSA is that we can fairly easily generate programs that detect various abstract patterns. For example, suppose we wanted to search of the pattern "[T|A]TT" where by [T|A] I mean T or A. Here is the FSA graph:

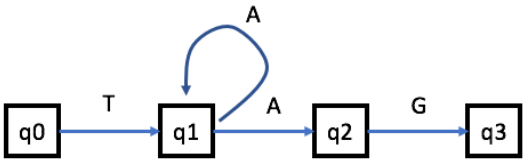


From **q0**, if the FSA sees the input "T" or "A" it makes the transition to state **q1**. We can interpret the state **q1** as the state corresponding to "I have seen one A or one T". Note the other arrows labeled with "A" that feed into state **q1**. For example, at state **q2**, if the input is "T", then the FSA progresses **q3**. If the input is "C" or "G", then it goes back to **q0**. But, if the input is "A", then this corresponds to "I have seen one A or one T" so it goes to **q1**. Other transition arrows can be figured out with similar reasoning.

Things get more tricky when we are trying to process patterns like "`[T|A]T[T|C]`".



Now we need to have two accepting states, **q3** and **q4**. I leave it up to you to trace out the logic of the above graph. You can try tracing the computation for some example strings to see that it works.



Finally, we might want to search for patterns like "`TA*G`", where by `A*` we mean one or more `A`'s.

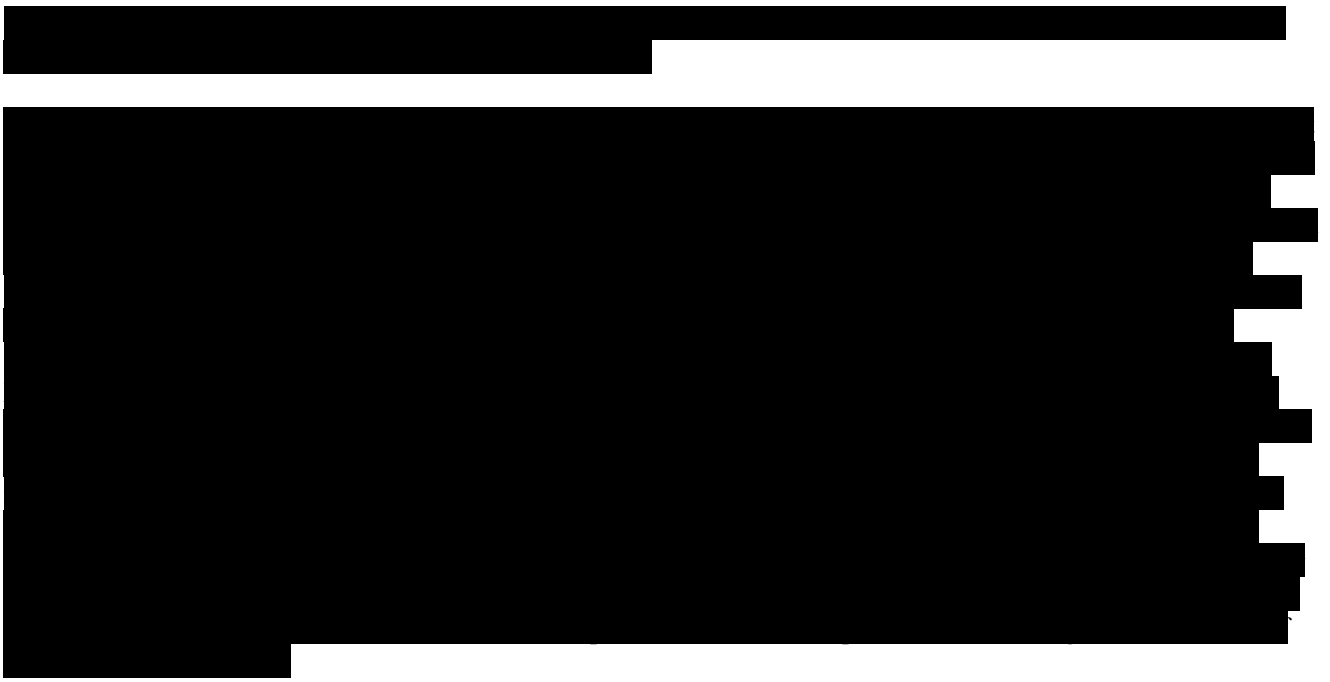
This is where we consider an important generalization of FSA. Consider the figure on the left. When we are in **q1** (the state corresponding to "I have seen one T") and we see the input "**A**", we move on to state **q2**. But, we also have the arrow back from **q2** that is also labeled with "**A**". This arrow tells us that when we are in **q1** and receive the input "`A`", the state of the machine changes to both **q2** *and* **q1**. That is, now the machine has two simultaneous states. When we follow the logic of this machine, we see that the machine will continue to keep track of inputs that look like "`TA*`" until the next input is not "`A`". When the transition tables of automata contain entries that tell the machines to make transitions to more than one state, we consider such events to be **non-deterministic** in the sense that more than one computational paths are generated. The above diagram shows a non-deterministic FSA (nFSA). There are several different ways to think about what it means to have a non-deterministic computation machine. One way is to imagine that when the machine reaches a state and input that results in transition to two different states, we now have two machines: one that follows the one path (say, transition to **q2** for the nFSA shown above) and another that goes to the other path (say, **q1**). Each time multiple state transitions happen, we give birth to new machines like a Sci-Fi multi-universe scenario. Each of our machines continues computing, and when any one of them reach the accepting state then we know we have found what we were looking for. So, a non-deterministic automaton can be seen a set of (possibly) infinite computing machines. Later, when we talk about the computational complexity of non-deterministic automata, we have to assume that possibly an infinite number of machines are doing the computing.

It should be obvious to you that we could have also created a FSA without two edges for the input "**A**". It will turn out that this is always the case for FSA; there is a deterministic equivalent version of a nFSA—but, a lot of times giving ourselves the option to use these empty edges gives a cleaner "program". If the automaton is a FSA, we don't necessarily have infinite number of computing machines even in the non-deterministic version. An FSA has no memory and has only its states and input symbols. Therefore, if there are finite states and finite symbols,

then for the nFSA there are only a finitely number of different combinations of states and input symbols. Any two machines in the same state receiving the same input symbol do exactly the same thing, thus, there is no need to consider the two machines as something different. For the nFSA processing "TA*G" shown above, at any step in the program execution, there are four possible states: **q0—q3**. If an input symbol and the state cause the transition to two different states, then we just keep track of the fact that we now have two machines. In general, for nFSA if we have **q** number of states, we only need to keep track of at most **q** states at any given time. What about computational time complexity? The transition table has the size |S|x|A|, where S is the set of states and A is the set of symbols (the alphabet). In general, to program a FSA to recognize a string of length *m,* we need |S| ~*O(m)*; that is, the size of the FSA program is upper bounded by the length of the string. So, constructing the transition table will require *O(mp)* time where *m* is the length of the string and *p* is the number of symbols in the alphabet set. Once we have such a transition table, we have to process each symbol of the input database string. So, if the length of the database string is *n*, then the computational time complexity is *O(n)* for the deterministic FSA. For the nFSA, we may have to keep track of up to *q* states at any input step, where $q = O(m)$ as we already discussed. Therefore, the computational time complexity will be *O(nm)*.

Automatons more complex than a FSA have the capacity to store information as well as change states based on input. For example, a full Turing Machine can be seen as having two sets of memory in a stack data structure. The state of the machine involves both the state of the automaton and the state of the memory. For a non-deterministic automaton, when we transition into multiple machines we have to keep track of all of their states. For FSA there were only a finite number of different states. But, for automatons with memory the state of the memory may have infinite possible configurations. Therefore, something more complex like a non-deterministic Turing Machine is equivalent to (possibly) an infinite number of deterministic Turing Machines.

Finally, we stated that FSA has a connection to formal grammar constructs called regular grammar. Formal grammars are rules for generating strings of symbols as we discussed previously when we discussed "languages". Formal grammars, as opposed to grammars for natural languages, have fixed rules by which "grammatically correct" strings of a language can be generated. There is a kind of a duality between automata that process strings of symbols and formal grammars that generate strings of symbols. An automaton will tell you when an (input) string contains strings or substrings that belong to some rule—that is, belong to a language. If a string is found to belong to a language by an automaton we say the string has been **parsed** by the automaton. So, an automaton for a given language can scan through **A*** (remember the Kleene star of the alphabet set **A**) and then tell you which ones are part of the language. A formal grammar gives finite rules by which all such strings can be generated by repeated application of the rules. For example, we may have the rule R -> aR to mean that we take some string represented by the variable R and replace it with another string that concatenates the symbol "a' in front of R. There is a one-to-one correspondence between a set of formal grammar rules called **regular grammar** production rules and a FSA instance. We won't dig too deep here, but these kinds of ideas have been adopted in computational biology to form the basis for modeling families of genomic strings. For example, we might want to have a set of grammatical rules or (conversely) a FSA that will generate/parse all amino-acid strings that belong to transcription factor family. [This is a hypothetical example, so far, we know of no such grammar.]



The distinction between computational complexity of an algorithm versus the complexity of a problem is a fundamental distinction in theory of computation. Known algorithms for a given problem will establish upper bounds on the complexity of a problem—after all, a problem can't be more complex than a known solution. Sometimes things like the basic need to read the input can set lower bounds on the problem. For any problem, when we have upper and lower bounds, we would like those bounds to be tight, in the sense that the actual behavior is close to the known bounds. Stating that the exact string search problem is $\Omega(1)$ (lower bound by constant time) and $O(2^{nm})$ (exponential in nm) doesn't really give much information. Proving a good set of asymptotic bounds for a problem (as opposed to an algorithm) can be very difficult. Consider the common grade school algorithm for multiplying two n-digit numbers in decimal notation. We all learned the algorithm of stacking the two numbers together, starting with the right hand most digit of the second number and then sequentially carrying out position-by-position single-digit multiplication. (We know how to do single-digit multiplication because we've memorized it; i.e., we have a look up table for it.) When all is said and done, if you look at the operations, you will see that the standard positional algorithm takes $O(n^2)$ operations, where n is the number of digits for both numbers. This is why we didn't like doing multi-digit multiplications! So, the problem of n-digit multiplication has a known $O(n^2)$ complexity bound. It turns out we can do better by dividing the problem with the strategy of splitting each number into two pieces. For example, if the two numbers are 372643 and 182737, we restate them as the numbers (372*1000+643) and (182*1000+737). You can think about this strategy and how it might help with the multiplication problem. I won't give the details here but I should note that this kind of approach, where we divide the problem into smaller pieces that can be solved more efficiently and then putting it back together is called (naturally) "**divide-and-conquer**." With this divide-and-conquer approach, it turns out that we can asymptotically approach $O(n)$ complexity for n-digit multiplications. Since reading the two digits takes at least n operations, we might say the problem has $\Omega(n)$ lower bound; and putting the upper and lower bounds together suggests that the problem of n-digit decimal multiplication has $\theta(n)$ time complexity.

Hamiltonian path/cycle problem: Given a graph G = {V, E}, find a path/cycle through the graph such that every vertex is visited exactly once. (Of course, the decision version of the problem will be "Given a graph G, is there a path/cycle…")

While seemingly an innocent sister to the Eulerian path/cycle problem, it turns out that there is no known polynomial time algorithm to decide whether a graph has or does not have a Hamiltonian path/cycle. To take a very naïve approach to the problem, if the Hamiltonian path/cycle exists, it must be a permutation of the list of the vertices. Suppose $|V| = n$, i.e., there are n vertices. Then, examining every permutation of the vertices will involve $n!$ possible cases. If we look up Stirling's approximation for the factorial function, we get:

$$n! \sim \sqrt{2\pi n} \cdot e^{-n} n^n$$

Or, $O(n^n) \sim O(2^{nlogn})$. This is a lot. Actually, we don't really have to look at every *n!* permutation of the list of vertices since most of these permutations will contain a sequence of vertices that are not connected in G. We can refine our analysis by considering the possible paths starting at one of the *n* vertices. Then, from the starting vertex, we will have p_0 possible next vertices, where p_0 is the degree of the starting vertex. Suppose we go to one of the secondary vertices, then we would have p_1 next possible vertices, where p_1 is now the degree of the second vertex, etc. As we expand out into possible paths, some of the path will be closed off because we cannot go any more or, we are forced to visit a vertex we have already visited. It is easy to see that we would have $O(nk^{n-1}) \sim O(2^n)$ upper bound on the possible paths where *k* is the maximum degree of a vertex in the graph. Whether we have $O(2^{nlogn})$ complexity or $O(2^n)$ complexity, the time complexity here is generally of the form $O(2^{poly(n)})$, where *poly(n)* denotes "polynomial function in *n*". We note for computational complexity statements, we will include functions of *log[poly(n)]* to be included in the polynomial function class, since such functions are upper bounded by polynomial functions of *n*. We call this the **EXP class** (of time complexity).

Following from the above accounting, the naïve *algorithm* for deciding the Hamiltonian path/cycle problem has EXP time complexity. We are not sure if the Hamiltonian path/cycle *problem* itself belongs in EXP time complexity class. There might be an efficient algorithm that avoids all the exhaustive enumeration of possible paths. But, as of now, we don't know any algorithm for the Hamilitonian path/cycle that does not have exponential complexity. Yet, here is an interesting twist to this problem. Suppose there is a computational wizard (sometimes called the *Prover*) who shows you (the *Verifier*) a putative Hamiltonian path/cycle. It is a simple matter (i.e., a polynomial process) to check the validity of the asserted path/cycle solution. A putative Hamiltonian path/cycle for a graph is something we call a *witness* to the fact that the graph has a Hamiltonian path. And, the validity of the witness can be checked with a polynomial time algorithm. In fact, we can check the witness with *O(n)* time complexity, where *n* is the number of vertices. Here is an analogy. Suppose somebody asks you to solve an algebraic equation for real-valued solutions. As you might remember from high school, finding solutions for equations can be hard and tedious. But, if somebody gives you some numbers and claims those numbers are solutions to the algebraic equation, you can easily plug in the answers and check to see that the putative solutions work. Those numbers are witness to the fact that the equation has solutions.

To reiterate, what we just said is:

Given some problem, if somebody gives us a possible solution to the problem, we can check in polynomial time if the solution is correct,

…and therefore verify that the problem can be solved. What is the big deal about this? It turns out that there are problems where this is not the case. Consider the standard chess game. Suppose there is a game in progress and we want to know "Is there a sequence of moves to guarantee that White will win?" Now, a computational wizard tells us that White will checkmate in 23 moves. If this statement is true, it is a witness to the original question and we would answer "Yes, there is a sequence of moves to guarantee that White will win." However, as you might imagine, it is no trivial matter to check the statement that there is a strategy for White to achieve checkmate in 23 moves. To see this, suppose that were to explain to you how there is a checkmate in 23 moves. I might start with, "I move my knight to position X", "then you might move your queen to X*", "or move your pawn to X**", "or ….", etc. That is, I have to describe all the possible events before I get to the 23rd move that puts the black in a checkmate. In fact, just describing this sequence of solutions will require exponential amount of time. Thus, there are decision problems, where somebody can give a plausible witness statement, yet, there is no polynomial time procedure to check the witness itself.

The discussion on the Eulerian path/cycle problem, Hamiltonian path/cycle problem, and the checkmate problem, suggests that regardless of the particular algorithm, these problems have very different complexity flavors and should be treated differently. We already know that the Eulerian path/cycle problem is in the complexity class P, since we know an algorithm that can solve the problem in polynomial time. A possible witness for the Hamiltonian Path/Cycle problem for a graph of *n* vertices is an ordered list of vertices, 1>3>2>4>…>…> *n*, that is a candidate Hamiltonian path/cycle. We can check this by ensuring that each vertex is represented once in the list and for every adjacent pair of vertices in the list that there is a connecting

edge. Therefore, this witness can be checked in *O(n)* time. For the chess problem, to give a witness to the fact that there is a guaranteed checkmate for white in, say 23 moves, we will have to first show a possible white move, then list all the possible black moves, then show all the white moves corresponding to each of the black moves, etc. That is, we will need at least an exponential sized witness, which will take at least exponential time to check.

It turns out that the Hamiltonian path/cycle problem is in a class called **NP**, which stand for non-deterministic polynomial. The NP class is characterized by the fact that a witness for the problem can be checked in polynomial time. The key notion for problems in the NP class is that a non-deterministic computational machine can solve NP class problems in polynomial time. Recall the discussion from non-deterministic FSA. A view on such machines was a computing machine whose states can multifurcate to generate virtual copies and simultaneously carry out multiple computing paths. If any one of the machines finishes, the problem is solved. If we consider the Hamiltonian path problem, we might consider a process that starts at some vertex and then considers all possible next vertices simultaneously, then again consider all the possible next-next vertices, etc. Each of these paths creates a new "virtual machine" and eventually one of them can verify a witness in polynomial time—as long as the path itself is polynomial in length. Thus, if we have a non-deterministic computing machine, we would be able to solve the problem in polynomial time. Of course, considered this way $P \subseteq NP$; every polynomial time algorithm can also be solved in polynomial time by a non-deterministic machine. The curious thing is that we can't seem to prove that this inclusion is proper. That is, we do not know whether there exist problems that are in NP but not in P. One of the most famous problems in computer science and mathematics is whether P = NP. There are absolutely strong reasons to think that P ≠ NP. We believe strongly that there are problems that are known to be NP and have no polynomial time solutions. But, no formal proof has been given despite many extremely gifted people working on it. In practice, we know a lot of problems for which no polynomial time algorithm is known, such as the phylogeny problem or the multiple alignment problem that we will discuss later; but we don't have a proof that polynomial time algorithms do not exist for these problems. Unfortunately, it turns out that many, if not most problems, in computational biology are in NP or even harder.

AA -> 2	**AC -> 3**	**AG -> 5**	**AT -> 7**
CA -> 11	CC -> 13	CG -> 17	CT-> 19
GA -> 23	GC -> 29	GG-> 31	GT -> 37
TA -> 41	TC -> 43	TG -> 47	TT -> 53

Unit 5: Large-scale searches and Re-sequencing Alignment

We now turn to the problem of doing very large-scale searches including aligning sequences that we know are not too different from each other. The most common application occurs in resequencing. In the resequencing problem, we already know the reference genome sequence, we have resequenced minor variants of the reference genome, and we would like to align the resequenced data to the reference genome. For the most part, in this kind of situation, we do not have to consider somewhat complex problem of positional homology problem that we discuss in the next unit (briefly, the problem of identifying positions in two or more strings that are related to each other by genetic descent). In the resequencing problem, we expect the two strings to be aligned to be nearly identical to each other with obvious positional homology. In this case, the alignment problem is really a string search problem for strings with some bounded difference—say different up to k positions. We could use any of the previous string search methods that allows us to search with up to k number of different letters. The key challenge is for the cases when we have a very large input size. We might have a large input size because the database we want to consider is very large (e.g., Genbank) or because the strings we want to align to the database is large (e.g., next generation sequencing). For these cases, we need to consider strategies that are better than $O(n)$, where n might be the size of the database string. After all, if we have 500 million 100-bp short reads of the human genome, which is 3 billion bases, we don't want to do 500 million searches at O(3 billion) time complexity. As we discussed in the section on prefix/suffix trees, when carrying out multiple searches (e.g., the short reads) over the same database string (e.g., the reference genome) it is usually best to structure the database string. The main problem is how to structure the database string when the number of searches or the database size is very large. Here we discuss several strategies such as indexing and filtration (also called exclusion method) and then look in detail at a space-efficient version of prefix/suffix tree.

We first start with the term "index." An index is kind of a handle into another object; or, more abstractly a function from the set of index values to the set of data objects. For example, an index of a book is a list of words and the pages of their occurrence. RAM memory of a computer is indexed by "addresses" which are some finite sized (binary) numbers and the memory locations contain various data objects. Programing languages allow us to map some variable we might write to the memory by these address indexes. For example, if I use an array and refer to X[1], X[2], etc., the compiler turns these into references for actual memory locations. In general, indexes are well-organized, say non-negative integers, and the object that they reference are more scattered, say all the different things we might put into memory. Thus, an indexing scheme gives us an ordered way to "handle" heterogeneous collection of objects by associating ordered objects to unordered objects. Finally, an indexing scheme might involve an algorithmic procedure that creates the association such as the FM-index we discuss below for Burrows-Wheeler Transforms.

Programming languages usually allow us to use various non-negative integer values as indexes of array variables and the compiler knows how to associate the memory locations. Which memory locations to associate for arrays is relatively straightforward because when the index is non-negative integers, association of the variables to the memory addresses is predictable. For example, a real-valued array running from index 0 to 1,000 can be associated to 1,001 consecutive 32-bit locations. We can think of this as taking one set of well-ordered numbers, the array indexes, and mapping it into another set of well-ordered numbers, the memory addresses. Now, let's consider a more general situation where we might want to store a genomic string in memory and then retrieve positions in the genome by using substrings as indexes. To do this, we need to take the general collection of things that we want to use as indexes, say substrings, and map them to numbers, which then can be mapped to memory locations.

Hashing is the term used to describe various systems to generate a relationship between a general set (e.g., substrings) and the set of non-negative integers with computational efficiency close to O(1) time; i.e., (almost) like RAM memory. We call such a map, a **hash function**. Let **S** be a set of objects of interest; for example, **S** might be the collection of short sequence reads from next generation sequencing. A (perfect) hash function $h()$ is a function from **S** to the natural numbers such that if $a, b \in \mathbf{S}$ then $h(a) = h(b)$, if and only if $a = b$. The hash function can be used to use elements of **S** to reference locations in the memory. For example, if we have RAM that is addressed by natural numbers, we might do something like $mem(h(a)) \leftarrow a$; that is compute $h(a)$ and then

put the object *a* into the memory location *h(a)*. Or, in general reserve *mem(h(a))* as a place to put information associated with the object *a*; this is called a hash table. An associative array that we discussed in data structures is an instance of this kind of procedure. For example, we might assign binary values to *mem(h(a))* to indicate whether sequences from a particular person contains the substring *a*. Then when we have a second set of sequences from another person, we can just examine *mem(h(a))* to see if *a* found in this person was also in the genome of the first person. Such a test could be done in O(1) (i.e., constant) time.

Suppose we have the reference human genome and a bunch of 100bp short reads from next generation sequencing. In the ideal case, we might break up the reference genome into **S** = {set of all possible 100bp substrings of the reference genome}, construct a hash table with the location of the 100bp substrings, then use the table to align the next generation sequence reads to the reference genome in constant time for each read. This means that we have a method of taking a 100bp string of {A, C, G, T} symbols and converting it into a number. One possible scheme is to encode each symbol as a number: {A→ 0, C → 1, G → 2, T → 3} and then the 100bp string can be represented as a concatenation of these numbers; i.e., each position in the string is encoded by a decimal shift. Or, a bit more compactly we could use binary numbers to encode the symbols, needing 2-bits per base position and then shift in binary positions. Such a system would take up 200-bit memory space, which is $2^{200} \sim 10^{20}$—a totally impractical amount of memory. In practice, we do not have infinite amount of memory, so the range of the hash function must be limited, say 0 to *v*, where *v* is the size of the total RAM. If $|\mathbf{S}| > v$, then we will no longer have a one-to-one relationship. In general, $|\mathbf{S}|$ may not be ungodly large. For example, we might have the human genome string of length 3 billion and we want to create an index to all k-mer substring locations. This will be at max 3 billion such k-mers and probably considerably smaller if k is not too large.

Constructing a good hash function involves trying to find a way to map **S** to a bounded range of number [0 to *m*], such that *m* is as small as possible, we can compute the relationship efficiently, and we avoid as much as possible having more than one element of **S** mapping to the same number. When two or more different elements of **S** map to the same number, we say that the hash function has a "collision". There are many different ways to resolve such a collision, usually involving examining the details of the object after initial hash function computation that results in using more computing time. A good hash function is difficult to construct. Many programming languages come with a generic hash function (e.g., Perl's associative array) but such functions are not necessarily very efficient (in memory and time). A more efficient hash function can be constructed for specialized applications such as genomic substrings, but this requires work beyond the scope of this text. For this course, we will just assume that if the size of **S** is not too large, we have efficient hash functions available. One last thought on hash functions is that there is a close relationship between the idea of hashing and the idea of encryption. When we encrypt some text, one way to think about this is that we are taking finite predictable numbers (think of letters as having numerical representations) and trying to map them to random (nearly) infinite range numbers. A hash function takes a collection that might be random and very large, say all possible substrings of a genome, and map it to smaller finite predictable numbers.

In general, if we were to take a human genome and break it into all possible 100 bp substrings, the size of the collection could be very large, up to 3 billion (imagine just taking overlapping 100 bp pieces). But, 3 billion is something that would fit into the RAM memory of modern computers so an appropriate hash function can be used to store the human genome using an indexing system of 100 bp nucleotide strings. However, since we would like to allow for some variations, we would have to hash a much larger collection than just the genome. If we were to allow up to two mismatches in each 100bp string, we now need 16x3 billion indices, suddenly not so easy to fit into standard computing hardware. Once we allow for more reasonable variations like indels, the size of the possible index space becomes impossible. Therefore, direct hashing of short 100 bp reads even with efficient ways to resolve collisions is not a practical solution.

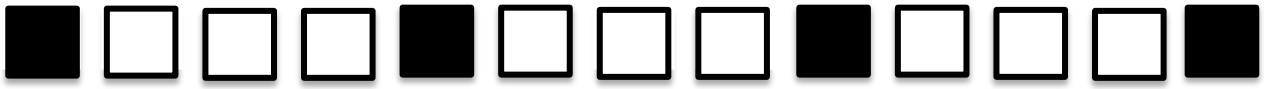
One possible approach is to consider a smaller subset of information about the genomic string, such as *p*-mers of certain size. For example, the genome may be "indexed" by a set of all possible 10-mers, where each possible 10-mer would be associated with a list of locations of those 10-mers. There are approximately 1 million all possible 10-mers of ACGT. Using 2-bits for each letter, an index of all possible *p*-mers can be stored in $4^p/4$ bytes (= 2^{2p-2}

bytes). Current computers easily have access to 8GB or so of memory so we can store an index of 17-mer or so in RAM. Longer the index length, smaller the number of possible positions in the genome, but more memory we need.

Of course, any particular genome like the human genome may not contain all possible *p*-mers, if *p* is of a reasonable size. So an index of all possible 17-mer is not likely to be space efficient—in those cases, we might use a hash function to reduce memory use by only mapping the 17-mers that appear in the reference genome. Suppose we now have a 100bp read and would like to find an exact match in the human genome. We can examine the first 17 bases of the 100bp read and then look up the index of 17-mers, which has the information on the positions where we can find the exact 17-mer. Then, we can extend the strings from the position of each of the 17-mer locations to see if we can find a match to the 100 bp read string. This is called filtration or exclusion. We used the index to filter out or exclude all the parts of the reference genome that cannot contain the 100 bp read. Filtration efficiency can be defined as the expected number of potential matches divided by the expected number of index match positions. For example, suppose for a random 100 bp read string, the expected number of match positions in the reference genome is 5 and the expected number of 17-mer indexed positions is 50, we would have a filtration efficiency of 1/10. Obviously, if we had a100-mer index, filtration efficiency would be 1. Various different schemes have been considered to increase the filtration efficiency without increasing the size of the index. In principle, a 17-mer occurs once every 16 billion bases for a random sequence with equi-probable bases. Thus, every 17-mer should be expected to be unique within a human genome and this kind of indexing should give us a very efficient alignment. Unfortunately, the human genome is highly redundant. More than 40% of the genome is comprised of nearly identical repeats of short strings. Thus, the filtration efficiency of any reasonable sized *k*-mer will not be very high.

As mentioned above, indexing exact substrings is not enough because we would like to find matches with *k*-mismatches rather than an exact match. Suppose our query string is size *m*, then an immediate fact is that the query string and the database string will share an exact match of a substring of size at least $l = \lfloor m/(k+1) \rfloor$ where $\lfloor x \rfloor$ means the integer less or equal to *x*. Thus, if we have 100 bp sequence reads and would like to allow up to 2 mismatches, we could filter by looking for exact matches of 33-mers. An index of 33-mers is not feasible, but when the read sizes were smaller (30 bp in the first next generation sequencers), this kind of filtration using a smaller *k*-mer was a feasible and efficient approach.

Is there a way to use reasonable sized *p*-mers for filtration but somehow avoid the problem with redundant substrings of the human genome? One idea is called gap filtration. In this strategy we create a *p*-mer that spans a length that is much larger than *p* by considering *p* positions, each separated by a gap of size *d*. See the figure below and note that *p* is the total number of positions that we are using while *p+(p-1)d* is the total span of the index:



The utility of a gapped index is two-fold. First, as mentioned, many biological sequences have positional redundancies in their sequence. An index is most useful if the occurrence of any index pattern is approximately random in the database sequence—otherwise, we might have some indices that map to tens of thousands of positions, greatly reducing efficiency. By placing gaps in the index, we can help avoid the redundancies that are often spatially adjacent. For example, a mammalian genome has hundreds of thousands (or even millions) of short repeated sequences. A gapped *p*-mer whose total span is greater than the typical size of the repeated sequences has a better chance of covering unique sequences that yields unique positions. Second, by using multiple gapped *p*-mer with different gap sizes, we can implement a multiple filtration procedure. In this strategy

Gapp

we would generate a series of gapped p-mers with different gap size d and then in our searches we can retrieve all positions that satisfy multiple gapped indices.

Suppose we had enough memory space to construct a p-mer table of a reference genome of size N and we wish to use this table to find a string of length m and let's also assume that $m/p = q$. Under the optimal scenario the probability of occurrence of any particular p-mer is 4^{-p}. Given some read, we might break it up into q pieces of p-mers and for any one of them we would expect $\sim 4^{-p}N$ number of "hits" in the human genome. One possible strategy is to use the p-mer with the smallest number of hits and then go to each one and see if we can extend it to the full m-length read string. This would take $O(4^{-p}Nm)$ time. As another possible approach, we could try to find the conjunction of the positions of the q set of p-mers. Suppose the genomic coordinates returned by the q different p-mers are denoted g_{ij} for the ith position of the jth p-mer. If the read string is in the genome, there must be a set of q different coordinates in the hit lists that are within m distance of each other (because the read string is length m). The naïve way to do this would be to consider the first two sets $G_1 = \{g_{ij} \mid j = 1\}$ and $G_2 = \{g_{ij} \mid j = 2\}$. Then for each coordinate in G_1 and G_2, we would try to assess which pairs are within p distance from each other (because adjacent p-mers in the read string should be p positions apart). Since $|G_1| \sim |G_2| = O(4^{-p}N)$, such pairwise comparison would take $O(4^{-2p}N^2)$ time. The list of joint hits is expected to be of size $O(4^{-2p}N)$ so the comparison with the 3rd list, G3 will take $O(4^{-3p}N^2)$ and the next step will be even smaller. Therefore, the time complexity of the naïve method will be determined by the first step and we can say the time complexity is $O(4^{-2p}N^2)$. If $4^{-p}N < m$ then this is better than just "hit and extend" method. The human genome is 3×10^9, so if $p > 13$ or so, then $4^{-p}N < m$. In fact, we can do much better than this. For example, if the coordinate lists are provided to us sorted from smallest to largest, we can do the conjunction search through G_1 and G_2 in $O(|G_1|+|G_2|) = O(4^{-p}N)$ time. All of this suggests that filtration can be a very effective strategy even if our available memory is limited. Well, obviously there has to be a caveat. The caveat is that because of redundancies of a typical genome (we also say the genome has *low complexity*) the expected occurrence of any given p-mer is not $4^{-p}N$. Some are much more prevalent than others. Sometimes we can potentially custom design various gapped p-mer indices whose occurrence is nearly uniformly random for a particular genome. This is one area of current algorithmic development.

Space efficient encoding of suffix/prefix strings

In general, if the number of mismatches we want to allow is small, the most efficient way to align the reads would be the prefix/suffix tree we discussed previously. Recall that once such a data structure is constructed, we can efficiently search in time proportional to the length of the query string. The problem was the potential space complexity of the tree, especially for small alphabet size like DNA strings. Recently, an approach using something called the Burrows-Wheeler Transform (BWT) was developed to help solve the space efficiency problem. The BWT is a representation of prefixes or suffixes of a string, which stores information corresponding to a suffix (prefix) tree in a data structure that only needs space in proportion to the size of the database string. It is also related a data structure called suffix (prefix) arrays. Accessing the information in the BWT requires a bit more work than moving down a tree graph but generally it can be accessed very efficiently. Here, I first describe BWT as a representation of suffixes of a string, then how we can use it to search for particular substrings.

BWT is a reversible transformation of the original (database) string into a new string (the transformed string). [I should note that in mathematical terminology a transformation is an one-to-one map over a set of things, such that we have an invertible relationship between pairs of objects in the set. For BWT the domain is the set of strings of length n.] It turns out that if we take a generic string and compute its BWT, the resulting string tends to group recurrent symbols together and therefore have better compression properties. Therefore, BWT has been used as part of compression algorithms. But, for substring searches, it is not too useful to think of it this way. Instead we can think of BWT as producing a string that encodes all the suffixes of a string in such a way that we can search through the suffixes from that string in an efficient manner. Suppose we are given the string "`ACAACGT`", then we can list all of its circular permutations (meaning shift each symbol ahead by one position, rotate the first symbol to the last position):

```
0:ACAACGT$
1:CAACGT$A
2:AACGT$AC
3:ACGT$ACA
4:CGT$ACAA
5:GT$ACAAC
6:T$ACAACG
7:$ACAACGT
```

Where the symbol "$" was used to denote the end of the sequence. I have also labeled the circular permutations from 0 to 7. (Yes, I started the index from zero, because this is how the algorithm is described in other literature and I didn't want you to get confused. Also, this makes it convenient to take into account the fact that we added a "$".) Another way to think about the circular permutations is that it is the list of all the suffixes of the string (i.e., think of the strings as ending at $) followed by the prefixes of the string. I will call the above list CP-table for Circular Permutation-table. Now we lexicographically (alphabetically) sort the strings (assume $ precedes other letters):

```
7:$ACAACGT
2:AACGT$AC
0:ACAACGT$
3:ACGT$ACA
1:CAACGT$A
4:CGT$ACAA
5:GT$ACAAC
6:T$ACAACG
```

This will be called **sorted** CP-table. The last column of the sorted CP-table is defined as the BWT of the input string. That is, "TC$AAACG" is the BWT of "ACAACGT$". The first column of numbers of the sorted CP-tables is the row indices of the original CP-table. It also happens to be the indices of the suffixes of the original string (see the suffixes in the original CP-table). The sequence of these suffix indices, (7, 2, 0, 3, 1, 4, 5, 6), is called the **suffix array** and keeps track of the original positions. What can we do with this BWT? First, very interestingly, this string "TC$AAACG" contains information to regenerate the entire set of sorted circular permutations. We can do this by the following steps:

1: Start with the BWT, which is the last column of the sorted CP-table:

```
T
C
$
A
A
A
C
G
```

2: Sort the BWT, which should be result in the first column, by definition (note that each column of the CP-table has all the symbols of the original string):

```
$
A
```

```
A
A
C
C
G
T
```

3: Add back the BWT

```
T$
CA
$A
AA
AC
AC
CG
GT
```

4: Sort the augmented columns, this generates the first two columns

```
$A
AA
AC
AC
CA
CG
GT
T$
```

5: Add back the BWT and sort and continue:

T$A	$AC	T$AC	$ACA	T$ACA	$ACAA
CAA	AAC	CAAC	AACG	CAACG	AACGT
$AC	ACA	$ACA	ACAA	$ACAA	ACAAC
AAC	ACG	AACG	ACGT	AACGT	ACGT$
ACA	CAA	ACAA	CAAC	ACAAC	CAACG
ACG	CGT	ACGT	CGT$	ACGT$	CGT$A
CGT	GT$	CGT$	GT$A	CGT$A	GT$AC
GT$	T$A	GT$A	T$AC	GT$AC	T$ACA

As can be seen in the above sequence, we continue to recreate the columns of the sorted CP-table by this sequence. Once we have the full CP-table, any one of the rows can be used to recover the original string. Thus, BWT is a reversible transformation. Why does the method work? (Hint, for any string S in the CP-table, S[1] is the symbol that immediately follows the S[n] symbol in the original string; or vice-versa, S[n] is the symbol that immediately precedes the S[1].)

The ability to reconstruct the input string from the BWT string is not so much important for our purposes, but it demonstrates the idea that the BWT string encodes information about the suffixes. Going back to the sorted CP-table:

```
7:$ACAACGT
2:AACGT$AC
0:ACAACGT$
```

```
3:ACGT$ACA
1:CAACGT$A
4:CGT$ACAA
5:GT$ACAAC
6:T$ACAACG
```

Suppose we want to search for some particular string, say "CAAC". Then the fact that the table shows all the suffixes of the original string means that if CAAC is a substring, there will be some string in some row that starts with CAAC (indicated by red letters in the above sorted CP-table). Furthermore, since the table is sorted, if there are multiple instances of CAAC, rows starting with that string will follow consecutively. Last, lexicographic order of the strings also means that we can find a particular substring by a search procedure that is sensitive to alphabetical order. Thus, the sorted table is like a suffix/prefix tree in the sense that the information in the substrings has been organized in an ordered manner. For example, in the sorted CP-table above, we see that AC is in the rows 2 and 3 (using 0 as the first row). If we read off the corresponding values in the suffix array (the first column of the above table) for rows 2 and 3, we get 0 and 3 (see blue numbers in the table). These numbers correspond to the offset position in the original string where we can find AC; that is, in the original string "ACAACGT" there is a substring AC in the 1+0 = 1st position and 1+3 = 4th position. We formalize this idea by defining two functions, min(**W**) and max(**W**) for some substring **W**, as the minimum and maximum row position in the sorted CP-table of the strings that starts with the substring **W**. Given the sorted CP-table, we would like to compute min(**W**) and max(**W**) for any query string. The suffix indices of the rows from min(**W**) to max(**W**) give us the information on the positions of the substrings we are looking for.

There are two problems here. First, the entire sorted circular permutation table is $O(n^2)$ in size, where n is the length of the database string. Therefore, as given, the CP-table uses a lot of space—as much space as the worst case in the suffix tree. Second, even if we were given the entire table and we use the fact that it is lexicographically ordered, it can take $O(n)$ time to find min(**W**) and max(**W**) by a naïve search. If the table consisted of the human genome, we will have about a billion rows starting with the letter "A"—a lot of possibilities to wade through to find, say, "AC". Fortunately, there is an iterative process where we can compute min(**W**) and max(**W**) efficiently using only the BWT, not the full table.

As might be expected, the algorithm proceeds by building on solutions to shorter strings. Suppose we already know min(**W**) and max(**W**) for some substring **W**. We wish to compute min() and max() functions for the substring x**W**, where x is a symbol; that is, an extension of **W** by one symbol. If we know how to do this efficiently, we should be able sequentially find any substring. Let's start with an example: we know the position of **W** = "AC" in the sorted CP-table above and we want to look for A**W**; that is, "AAC". Given that the table is lexicographically sorted, wherever "AAC" is we know that its row position is greater than any rows that contain symbols that are lexicographically less than "A" (i.e., "$"). If we were looking for "GAC", then its position is greater than any rows that start with A or C and less than the rows that start with T. The first step in our algorithm is to use this kind of information to establish a baseline starting position for any string x**W**. Suppose we were looking for "CGT". Then since it starts with "C", we know its row position has to be between [number of A's]+1 and [number of A's] + [number of Cs]. In the original string "ACAACGT", there are 3 A's and 2 C's so "CGT" is in between row indices 4 and 5. We create a function N(x) as the number of symbols in the database string that is lexicographically less than the letter x (not counting $). So, in the string for the CP-table above, we will have N(A) = 0, N(C) = 3, N(G) = 5, and N(T) = 6. This function allows us to know that any string "Axxxx" will be within the row range 1 to 3, "Cxxx" will be within the range 4 to 5, etc. The function N(x) basically tells us the starting row of the extension x**W**, since the substring x**W** has to start after all the rows of the symbols that are less than x (in lexicographic order).

Assume that we know min(AC) = 2 and max(AC) = 3. We would like to use this to find the offset of xAC amongst the rows that start with the symbol x. We can first use N(x) to tells us the starting position. The pattern xAC is somewhere after N(x). We want to compute this offset. This is where the properties of the CP-table become critical. Recall that for any row of the circular permutation, S[n] is the letter that immediately precedes S[1]. So,

if xAC exists as a substring in the original string, there will be rows in the CP-table whose strings have the form "AC…x"; that is, strings that start with the substring AC and end with the symbol x. In general, suppose we have x**W** then there will be strings of the form "**W**….x" (i.e., starts with W and end with x) somewhere in the CP-table. Suppose we count the strings in the CP-table of the form "**W**….x" with **W** < AC. Then, this also counts all the strings of the form x**W** where **W** < AC; call this number *g*. Recall that N(x)+1 is the starting row index of all strings that start with the symbol x. Then N(x) + *g* + 1 is the starting row index of all circular permutation strings of the form "xAC…". If we also count the strings of the form "AC…x", call this number *h*, then N(x) + *g* + *h* is the last row index of all strings of the form "xAC…"

How do we figure out *g* and *h*? If we look at the row position of min(AC) (i.e., 2 in the table above) and count all the symbols in BWT (the last column of the sorted CP-table) that are x, but is less than the position 2, that corresponds exactly to counting all strings of the form "**W**….x" where **W** < AC. Thus, we can compute the minimum offset (i.e., *g*) of any string that starts with xAC. Then we add +1 for the actual position because xAC will start one position after this. For the maximum offset, we count strings of the form "**W**….x" where **W** ≤ AC; that is, we include AC. To do this, we look for incidences of x in the BWT that before the position of max(AC). In our example for AAC, the minimum offset can be obtained by counting the number of A's in the BWT before position 2. There are zero such A's. So, the offset of minimum position of AAC is +1 (= 0 + 1). For the maximum offset, we search for A's in positions equal to or less than 3. There is one of those. So the maximum offset is +1. The base position is 0, since there are no symbols less than A (except for the $ symbol). So we come to the conclusion that AAC can be found in the row range 1 to 1 of the sorted CP-table. We look up the corresponding suffix array value, which is 2, and declare that there exists AAC in position 2 of the original string. The key here is that we didn't actually need the full CP-table. We just did all of our computations from the last column, i.e., the BWT string. Thus, there is no need to keep the full CP-table with O(n^2) memory usage. The BWT string only uses O(n) memory.

To put this all together in a more precise form, let B[x, *i*] be the number of symbol x's found at or before the position *i* in the BWT string. Then we have the following recursive formula:

min(x**W**) = N(x) + B[x, min(**W**)-1] + 1
max(x**W**) = N(x) + B[x, max(**W**)] (Eq 1)

To fix ideas, let's compute the function values for AAC, recursively starting with "C", then "AC", then "AAC".

min(C) = N(C) + 1
max(C) = N(G)

For any single symbol X,

min(X) = N(X) + 1 and max(X) = N(Y) (Eq 2)

where Y is the symbol that lexicographically follows X. We can consider Eq 2 as the boundary conditions for the recursive formula of Eq 1.

For our string "ACAACGT" the BWT is "TC$AAACG", so we have:

min(C) = N(C) + 1 = 3 + 1 = 4
max(C) = N(G) = 5

Fig 2 shows that strings starting with "C" are indeed found between 4th and 5th row index. Before we go on, we can just pre-compute the function N(x) and B[x, i] (go back and recall the definitions of these functions):

```
7:$ACAACGT
2:AACGT$AC
0:ACAACGT$
3:ACGT$ACA
1:CAACGT$A
4:CGT$ACAA
5:GT$ACAAC
6:T$ACAACG
```

Fig 2

N(A)	N(C)	N(G)	N(T)
0	3	5	6

	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7
A	0	0	0	1	2	3	3	3
C	0	1	1	1	1	1	2	2
G	0	0	0	0	0	0	0	1
T	1	1	1	1	1	1	1	1

Now,

min(AC) = N(A) + B[A, min(C)-1] + 1 = 0 + 1 + 1 = 2
max(AC) = N(A) + B[A, max(C)] = 0 + 3 = 3

We check Fig 2 again. Next,

min(AAC) = N(A) + B[A, min(AC)-1] + 1 = 0 + 0 + 1 = 1
max(AAC) = N(A) + B[A, max(AC)] = 0 + 1 = 1

We are right again!

To find any substring using BWT, we need the suffix array (because it keeps track of the offset positions of the suffixes), we need the function N(x) and the function B[x,i]. N(x) is a simple function and can be computed in O(*n*) time and takes up space only in proportion to the size of the alphabet. B[x,i] is a function that takes up O(*kn*) space where *k* is the size of the alphabet and *n* is the length of the database. One complication is that if n is large, it can take a large amount of memory to represent the number. So, in fact, the B[x, i] table takes O(*knm*) space where *m* is the amount of space needed to represent the positions of the string. If the length of the string is *n*, then our positional notation for numbers takes log(*n*) space. That is, "1000000" denotes a million numbers but our notation for the number only takes 6 positions. So, the total size needed by B[] is O(*k nlog(n)*). We can also compute the table for this function in O(*n*) time once we have the BWT (we just need to count). The suffix array takes up O(*n log(n)*) space as well, where again log(*n*) is due to representation of the numbers. As one might imagine from the case of prefix/suffix tree, there exists an O(*n*) time algorithm to compute the BWT. The O(*n*) algorithm uses O(*n*) space to compute BWT. The human genome is on the order of 1 GB of bit-encoded storage (each of the four bases take 2-bits and each byte stores 4 letters) and most computers do not have more than 8 GB of memory, so the exact constant in front of O(*n*) matters. It turns out there are more space efficient algorithms that have time complexity O(*n log (n)*).

In sum, the BWT allows us to obtain a suffix array and a representation of the prefix/suffix tree that is space efficient. Searching through the BWT representation is a bit more involved than a simple search down a tree, but as you can see as long as N(x), B(x,i) functions are precomputed, it can be done in time proportional to the length of the query string. The BWT has been effectively used in many programs for next-generation sequence alignment. On the other hand, as the length of the query string goes up we may have problems related to handling *k*-mismatches. Under a model of uniform mismatch rates, the value of *k* will have to increase with the increasing length of the sequence reads. A naïve extension of the *k*-mismatch search on BWT will grow exponentially in the number of mismatches. Next generation sequencing is being used for other applications that may involve fairly large differences in the reference genome and the query string. For example, cancer cells may have a large number of insertions, deletions, duplications, and other genomic rearrangements. In fact, a key characteristic of cancer genomes is the presence of unusually large number of genomic rearrangements. Exact matching or even *k*-mismatch searches will fail to align sequences around genomic rearrangements. Next generation sequencing is also being used for RNA sequencing. RNA sequences have structural variations like splicing and RNA edits. In fact, RNA sequencing data is revealing that there might be extensive variations in the RNA transcripts that are considerably more different than previously thought. Thus, the RNA copy of the DNA information may be quite different in sequence from the genomic DNA sequence. In all of these cases with

structural variations, indels, or large number of mismatches, using a filtration strategy with an appropriate index set is likely to provide better sensitivity. A common procedure now is to use BWT type of efficient search to align reads and then take the left over reads that cannot be aligned in that manner and use a more compute intensive approach that couple filtration techniques with other approaches.

Unit 6: Sequence Alignments, Combinatorial Optimization

In this unit we consider the classic sequence alignment problem. In an alignment problem we are given two (or more) strings and we wish to match them to each other, the best we can. Therefore, you can think of this problem as a database-dependent search problem where the result and the algorithm involve optimizing the search over the database (i.e., the second string). This immediately tells us that the computational problem will be harder because we are forced to examine the whole database. There are two kinds of alignment problems. The first involves matching the strings with some notion of best, which must be modeled, while the second involves matching strings that are nearly identical and therefore the notion of best is very simple and problems are closer to the string search problems we discussed in previous units. For the first class of problems we have to work on both the modeling problem and the computational problem. For the second, which typically arises in next-generation sequencing applications, the main problem is the computational problem, which we already discussed in Unit 5—but, we note that the kinds of large-scale searches described in Unit 5 is also trying to solve a version of the alignment problem, therefore, the general discussion on the alignment model also applies to that class of problems.

We first define sequence alignments (we will generalize the definition later). Let S and P be two strings. We consider augmenting the strings by adding a "-" (gap) symbol at various positions. For example, if S = "AACCG", augmented S' might be "--AAC-CG". A sequence alignment of S and P are two augmented strings S' and P', such that |S'| = |P'|; that is, the augmented strings have the same length. In this manner, there is an invertible map between the positions of S' and P' where S'[i] ~ P'[i] are ***aligned***. Here is an example of possible alignments for S = "ATTTGGAA" and P = "TTGCA".

```
S′= ATTTGGAA
P′= -TT-GCA-

S′= ATTTGGAA
P′= --TTGC-A

S′= ---ATTTGGAA-
P′= TTGC-----A--

S′= -----ATTTGGAA
P′= TTGCA--------
```

The third alignment above has a "-" in the last position of both S' and P'. In general, we consider such aligned positions of two gap symbols to be superfluous, so in most alignments both the 1st position and the last position have at least one symbol from the original S and P. The fourth alignment above shows that a valid possible alignment is one that offsets the original strings completely. The main utility of alignments is that there is supposed to be biological meaning to the positional correspondence S[i] ~ P[i]. The simplest such meaning arises when S might be a known genomic sequence and P is a string obtained from an instrument that sequenced the genome. Then the meaning of the correspondence S[i] ~ P[i] is clear: P[i] is the copy of S[i]. We might see cases of S[i] ≠ P[i] within some alignment, in which case, we might attribute this to sequencing error. However, in more general situations, S and P are related strings, but not supposed to be exact copies of each other. In most cases S and P are genetically related to each other by ancestor-descendent relationships. That is, S and P are genetic copies of some common ancestor in the past. Recall that all parent-child relationships involve copying the genome from one generation to another. In the process of this copying, alterations can happen in the child. The simplest kind of alteration would change the base identity, say a mutation from A to C. If we were to look at two different children of the same parent, one child might inherit the string S = "AATTGGAA" and another might inherit P = "CATTGGAA". While S[1] ≠ P[1] there is still a notion of correspondence between S[1] and P[1]. Namely, they are both (imperfect) copies of the mother's string, call it M, so that S[1] and P[1] are identical in their relative position in the genome due to common descent from M[1]. We call this "**positional homology**" and say that S[1] and P[1]'s are positionally homologous.

Homology is a very important concept. Given two objects, if some of their features are identical (e.g., relative position in a genomic string), and this identity is due to a shared common ancestor, we call the feature (or by implication the object with the feature) homologous. Homology is a logical concept denoting a relational map of biological measurements and/or objects. The shared features due to the shared common ancestor distinguishes the origin of common feature from that arising due to other factors such as chance or physical forces. It is important to note that two or more things are either homologous or not homologous. There is no concept of "percent homology". (One might have two objects that have certain percent similarity.) The biological goal of sequence alignment is to identify positional homologies between nucleotides or amino-acids. We need a model to make this happen.

We go back to the consideration of mother-child genome and think about the kind of things that can happen during the biological replication from the mother's string, M, to the child's string S. We already discussed the case of a single nucleotide mutating (a point mutation). Another common possibility is that some nucleotides are inserted into the sequence, say transforming M = "AATTGGAA" to S = "AATTTGGAA" or S = "AATTTGTGTGAA", etc. This insertion may be random, which can happen through the random mismatch repair mechanism. More commonly, the inserted sequence may come through duplication of nearby sequence, say transforming M = "AATTGGAA" to S = "AATTAATTGGAA" (called tandem duplications; we also call such nearby sequences *cis*- sequence). Or, we may get an insertion of a copy of a string from far away (we call this *trans*- sequence, in biology), say through retrotransposition. When a random fill-in happens, the inserted nucleotides have no "ancestor"—we can think of them as spontaneously generated. However, when local duplication occurs or some remote sequence is copied in, the ancestor-descendent relationship of the sequences become complicated; one part of the genome may be the ancestor to another part of the genome. Other kinds of changes can happen, albeit not commonly. We might have local inversion, say M = "ATTGGAA" to S = "ATTAAGG", typically due to the generation of loops in the DNA. Other possible changes are duplication and inversion; chromosomal translocations, where large pieces of the DNA become cut and pasted elsewhere; and other things. Of course, much more commonly, we can have deletions of pieces of the string. When we compare two strings, it is hard to tell whether one of the strings has an insertion or the other had a deletion, so we commonly call them **indels**.

The possible biological processes that change the child's sequence from the mother's sequence can be thought of as a kind of an **edit operation on strings**. Given two homologous strings, they are related to each other by the initial ancestral common sequence and then diverge from each other through a sequence of biological edit operations carried out on the strings throughout their respective genealogical history. (Of course, two strings may not have any common ancestor, in which case they are not homologous.) So, the first step in inferring positional homology of sequences is to model the edit operations. One possible model is some kind of a stochastic process model that specifies the probability of each kind of possible edits per generation. It turns out such probability models for the kinds of mutational edits we discussed are extremely complicated and hard to convert into an inference method. Instead, the standard approach is to specify an approximate notion of goodness-of-alignment with a numerical score and try to optimize this score. Modeling a problem as an optimization problem is an extremely common form of modeling and we will have more to say about this later. In this particular case, we will be modeling the alignment problem as a combinatorial optimization problem, where the term combinatorial refers to the fact that the objects that are being explored are sequence alignments, a kind of a "combinatorial object", as opposed to numerical optimization, which involve real-valued numbers. In optimization, we have some real-valued function, $f(X)$, where X is the object being evaluated (the **argument** to the function). For each X, the function $f(X)$ measures some kind of notion of goodness of X with respect to our goals; therefore, we call this $f(X)$ function, the **objective function** of the optimization. Choosing a particular objective function corresponds to choosing a particular model for the problem at hand (e.g., the problem of inferring positional homology).

There are many models for sequence alignments in the literature. Most models only allow two types of edits, point mutations and indels. Other types of edits (i.e., translocation) are very difficult to model or compute, although some algorithms do exist. Here, we will only consider the point mutation/indel type of edits under three different variations—the first two are simple models mostly to help us understand the problem and the

basic algorithm. The third model, called the affine gap penalty model, is the bread-and-butter of sequence alignments. We will discuss the algorithm for the affine gap penalty model and then consider some additional variations.

The objective function for sequence alignments takes as an input a potential alignment and computes a numerical score. For example, we might have these two possible alignments:

```
S′= ATTTGGAA
P′= -TT-GCA-

S′= ATTTGGA-A
P′= --TTG-CA-
```

One possible objective function (and therefore a model) is to consider the edits implied by the positional homologies of the alignment. I've colored such edits in red. The two alignments above. If we now give each edit a unit score, the two different alignments above would have the scores 4 and 6, respectively. This kind of a score (i.e., objective function) is called an **edit distance**. In this particular case, we would propose to minimize the edit distance score function, thereby minimizing the number of edits between the two sequences. This corresponds to the somewhat vague idea that, all things being equal, mutations/indel are rare and therefore any two related sequences should have a minimal number of edits between them.

Another possible objective function is to compute a notion of similarity between the aligned sequences. For the two alignments above, we might compute the number of positions that have the same nucleotides. The scores then will be 4 and 3, respectively. In this case, we would want to maximize the objective function because (again somewhat vaguely) we think if two sequences are related they should be as similar as possible. One advantage of a similarity score is we can start modeling the notion of similarity itself. For example, we might consider the following "cost" matrix C:

	A	**C**	**G**	**T**	**-**
A	1	-1	-1	-1	-2
C	-1	1	-1	-1	-2
G	-1	-1	1	-1	-2
T	-1	-1	-1	1	-2
-	-2	-2	-2	-2	undefined

Each element, *C(i,j)* gives a score for each position of an alignment as a function of the paired symbols in the position. Note that the symbols in the table also include the "**-**" symbol that denote an indel. In general, we would have a *k* by *k* cost matrix where *k* is the size of the alphabet set in our strings. The alignment

```
S′= ATTTGGAA
P′= -TT-GCA-
```

would have the score -2 + 1 + 1+ (-2) + 1 + (-1) + 1 + (-2) = -3. Using this cost matrix, we can model the score of an indel differently from a point mutation and we can also model the score of different kinds of point mutations differently. Thus, this class of objective functions gives us more flexibility in designing models for biological scenarios (e.g., certain kinds of mutations might be more common than others).

As mentioned the most common kind of alignment model involves an objective function called an affine gap penalty function. Here, the main idea is that biological insertions or deletions don't happen individually but can happen involving multiple base pairs, albeit longer indels might be less likely. We score the indel by a "gap penalty function" of the form:

$-(Wo + We \cdot h)$ [note the minus sign in front of the parenthesis]

where *Wo* is a number called the "gap opening penalty" and *We* is another number called the "gap extension penalty" and *h* is the size of the gap. For example, let *Wo* = 3 and *We* = 1. Now, consider the following alignment:

```
S' = ATTTGGA-A
P' = --TTG-CA-
```

There are two kinds of gaps, a single two-position gap (marked in blue) and three one-position gaps (marked in red). The two position gap would incur the penalty $-(3+1*2) = -5$, while the one position gap would incur the penalty $-(3+1*1) = -4$. The function is called affine gap penalty because it is an affine function of the length of the indels (i.e, the gap). In this model, we will also have to score the positions that have non-gap symbols on both sides such as the 3rd position (`T:T`) and the 7th position (`A:C`). (I will use the notation `X:Y` to refer to an aligned position with symbol `X` on the top and `Y` on the bottom.) For the, so-called "**match positions**" (the positions of the alignment which do not have gap symbols on either string), we can again specify a cost matrix similar to the one above (with one less row and column since we are treating the gaps separately).

Having chosen an inference model through the choice of an objective function, we now need to optimize it. This means considering all possible alignments of two strings, computing the objective function, and settling upon the alignment function that maximizes or minimizes the objective function. Unfortunately for us, the number of possible alignments of two strings S and P of length *n* and *m*, is rather large. We can write an exact formula for the number of possible alignments (how?) but suffice it to say that it is exponential in the lengths of *n* and *m*. Fortunately, the situation is not as bad as one of those NP problems. It turns out that sequence alignment problems have the property that global solutions can be built up from smaller solutions. The main idea is to reduce the problem to smaller pieces and extend the solutions, while guaranteeing that the extension leads to global optimum. [A quick word on local and global optima. Objective functions can have local optima. Local optima are like hills in a mountain range. The form peaks relative to nearby points, but not necessarily the tallest peak (which would be the global optimum).]

Suppose that we have two strings S and P, each of length *n* and *m*, respectively. We will consider the *i*th and *j*th prefixes of S and P. Since we will talk about this a lot, I will use the short-hand notation [*i, j*] to refer to the *i*th and *j*th prefixes of S and P. For example, suppose we want to align the string S = "`ACTTG`" with the string P = "`AACTCG`". Then the following are examples of the notation:

[1,1] → S = "`A`", P = "`A`"
[1,3] → S = "`A`", P = "`AAC`"
[3,1] → S = "`ACT`", P = "`A`"

We break up the problem of optimal alignment of S and P to the problem of optimal alignment of the *i*th and *j*th prefixes of the two strings; i.e., the best alignment for [*i, j*]. Note that the original problem is solved if we have the optimal alignment for [*n, m*]. To find optimal alignment for [*i, j*], we start with the assumption that we know all optimal alignment for any combination of shorter prefixes. For example, [*i-1, j*], [*i-2, j*], [*i-1, j-1*], and so on. Now in considering [*i, j*], we can ask whether we need the information from all the shorter alignments. The answer depends on the objective function model. In the case of the edit distance objective function and similarity objective function, it turns out that to find the optimal score for [*i, j*], we only need to consider the optimal scores for [*i-1, j*], [*i, j-1*], and [*i-1, j-1*] and how to extend the scores to [*i, j*] in an optimal manner. This is because the score assigned to each potential alignment position is independent of the other positions (e.g., we do not assign the score taking into account previous positions) and it is additive. Thus, we can take the strategy that we will optimally align [*i, j*] prefix pair by using one-position shorter alignments and then adding one more position to get to [*i, j*]. Since, there are two strings, we can have three different possibilities: String 1 is one position shorter,

String 2 is one position shorter, or both strings are one position shorter. That is, [*i-1, j*], [*i, j-1*], [*i-1, j-1*]. From each of these shorter alignments we build to the alignment of [*i, j*]. There are thee possibilities:

(1) From [*i-1, j*], we insert the *i*th letter from S and an opposing gap symbol to get the alignment of [*i, j*]
(2) From [*i, j-1*], we insert the *j*th letter from P and an opposing gap symbol to get the alignment of [*i, j*]
(3) From [*i-1, j-1*] we insert the *i*th letter from S against the *j*th letter from P to get the alignment of [*i, j*]

The cost of any of these three incremental steps can be read from the cost matrix (or the edit operation considerations). So, the possible score at [*i, j*] given the score of the shorter alignments are:

(1) Optimal score up to [*i-1, j*] + cost of *i*th letter from S aligned against a gap symbol
(2) Optimal score up to [*i, j-1*] + cost of *j*th letter from P aligned against a gap symbol
(3) Optimal score up to [*i-1, j-1*] + cost of *i*th letter from S aligned against the *j*th letter from P.

The optimal score at [*i, j*] is the optimal value amongst one of the (1), (2), or (3) possibilities. (Note, the "optimal" may be the maximum or the minimum depending on your objective function.)

This is the basis of ***Recursion Formulas*** for computing the alignment objective function. I'll just pick one for the similarity alignment for illustration (here C(x,y) is the cost matrix):

$$V(0,j) = \sum_{1 \le k \le j} C\big(-, P(k)\big)$$

$$V(i,0) = \sum_{1 \le k \le i} C(S(k), -)$$

$$V(i,j) = max \begin{cases} V(i-1, j-1) + C(S(i), P(j)) \\ V(i-1, j) + C(S(i), -) \\ V(i, j-1) + C(-, P(j)) \end{cases} \qquad \text{Eq 1}$$

The third formula for *V(i, j)* in Eq 1 expresses exactly what we discussed above. To solve *V(i, j)* we need to know some starting values; what we call **boundary conditions**. This is given by the formula for *V(0, j)* and *V(i, 0)*. The values for the boundary conditions follow common sense. Each corresponds to aligning prefixes of S or P against the empty string. The recursion formula succinctly describes the idea of finding optima for larger objects ([*i, j*] prefixes) from optimal solutions of smaller objects ([*i-1, j*], [*i, j-1*], [*i-1,j-1*] prefixes).

To carry out the computations, we can use a **recursion function** in a programming language to compute *V(i , j)* until we get *V(n, m)*—that is the optimal value for the full lengths of S and P. It turns out that using a recursion function is not the most efficient way to compute *V(i, j)*, but for principles sake let's discuss recursion a little bit since it is a common programming device.

BOX: Recursive Functions

One of the technical things you may have trouble digesting, if you've never seen this before, is recursive functions and recursive algorithms. The basic idea is simple. We can often define an operation as "similar" to a previous operation but with some additions. Another way to think about is as building a solution utilizing previous solutions. Eq 1 above is a kind of such iterative relationship. In fact, a mathematical series is the most basic example of defining new quantities or functions by previous quantities or functions. For example, let's consider the following definition:

$$F(n) = F(n-1) + 1, \quad n = 1,...,\infty$$
$$F(0) = 1 \qquad \text{(Eq 2)}$$

The first part of Eq 2 is a function whose argument is a non-zero integer. The second part $F(0) = 1$ gives the boundary conditions.

Eq 2 is easy to understand going forward in the index *n*. Here are some values:

n	*F(n-1)*	*F(n)*
0	Undefined	1
1	1	1+1 = 2
2	2	2+1 = 3
3	3	3+1 = 4
4	4	4+1 = 5

We see that in fact *F(n) = n+1*. It is a simple function that returns the argument plus one. In practice, we might like to know the value of F(10). Then by the definition given in Eq 2, we need to know F(9), which in turn requires the knowledge of F(8), etc. until we get to F(0). The value of F(0) is given by the boundary conditions so we can move back up to F(1), F(2), etc, until we obtain the value of F(10). The fact that the function recursively defines the values by previous values is the reason we call this a recursive function.

For Eq 2, we can solve the recursive formula to derive an algebraic formula that directly gives the value in a closed form. In other cases, it is not so simple. For example, consider the Fibonacci series defined as

$$F(n) = F(n-1) + F(n-2), n = 2,...,\infty$$
$$F(0) = 0, F(1) = 1 \qquad \text{(Eq 3)}$$

Can you write a closed form formula for Eq 3?

We now consider a program for computing *F(n)* in Eq 2. First, an algorithmic function is more or less just like a mathematical function. It takes input values and returns an output value. So we might write something like this:

```
Function F(A,B,C)
{
        Do some stuff              (Prog 1)
        Return Z
}
```

where A, B, and C are the arguments (= input) to the function and I wrote three of them just to emphasize that there might be multiple arguments. The syntax of Prog 1 should be pretty intuitive. It says "get as input A, B, C from external conditions", "Do things inside curly brackets", and "Return the value Z". The value Z might be a number or something more complex. In computer programming we construct functions like Prog 1 and then "call" them by writing their names down with a particular argument value. In other words, when we evaluate a program function *F(X)*, we also say we "call *F(X)*".

We now write Eq 2 as a program:

```
Function F(n)
{
```

```
        If(n<0) "Say I give up"
        If (n=0) Return 1
        If (n>0) Return F(n-1)+1
}                                   (Prog 2)
```

Now consider what happens if we call the function in Prog 2 with *F(3)*. That is, I want the function value when I pass the argument "3".

Step 1: 3 is not negative so I don't give up
Step 2: 3 is not 0 so I don't return 1
Step 3: I decrease 3 to 2 and then call the same function with the argument 2, i.e., call *F(2)*. I wait for the return value of *F(2)* and return that value + 1.

The recursive part of the function is in Step 3, where we call the same function but with a different argument. If we think about it, when *F(2)* is evaluated the same thing will happen and *F(1)* will be called. Only when *F(0)* is called will a concrete value (=1) be returned. So the value of *F(3)* is not known until we reach back to *F(0)*. After that *F(1)* then takes the *F(0)* return value, adds 1 and returns 2 to *F(2),* which then adds 1 and returns that value to *F(3),* etc. So, there is a hierarchical nesting of function calls and return of computed values. Concentrating on the values returned, we can rewrite what is happening like this:

$\text{Return}_3(\text{Return}_2\ (\text{Return}_1\ (\text{Return}_0\ 1)\ +\ 1)\ +1)+1$

Where the subscripts indicate the argument value for which each return value is being generated.

Recursion seems complicated when we think of this kind of nested hierarchical function evaluations. But, if we concentrate on the forward definition such as in Eq 2 and Eq 3, you can see that the actual construction is easy. Here is another example that computes *n!*

```
Function F(n)
{
        If(n<0) "Say I give up"
        If (n=0) Return 1
        If (n>0) Return F(n-1)*n
}                                   (Prog 3)
```

A recursive function need not have integers as arguments. Here is a program that takes in some positive real number and splits the value in half and computes some value. It keeps splitting until the intervals are smaller than a predetermined size. Therefore, the program returns how many halvings are required for the original value to become smaller than the predetermined size. I don't think there is any practical use for this function but please try to follow what it is doing as an exercise.

```
Function Split(X)
{
        if(X<0) "Say I give up"
        if(X< 0.0001) return 1
        if(X>= 0.0001) return F(X/2) + 1
}                                   (Prog 4)
```

Lastly, a recursive function or a recursive program is very nice because it often leads to a clean simple way to describe a complicated problem. For example, the closed form function for the Fibonacci series in Eq 2 is difficult to derive but the statement as a recursive function is simple. However, in terms of computation, there is a great cost to defining functions this way. As noted above, each function evaluation has to wait until all the "lesser" functions are evaluated before finishing. In a computer program all this waiting is stacked into a special memory called the "stack" (ta da! We already saw this before). If you called Prog 3 with a large argument value,

say F(1000000), that could cause a lot of memory to be used up in waiting. All recursive functions have equivalent definition in a non-recursive form, although that might not be easy to find. Efficient programmers try to avoid recursion unless the number of function calls is likely to be small.

Dynamic Programming to compute the alignment function

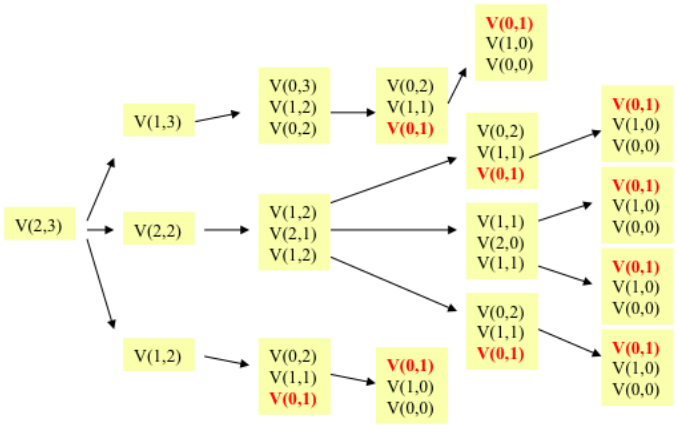
If we were to use standard recursion function to solve the recursive formulas in Eq 1 above, we might construct a function that looks like this (assume this function has access to the cost matrix *C(i , j)* and the two strings S and P):

```
Function V(i, j):

If( i = 0 and j = 0) return 0
Else if (i = 0)  return the sum of C(-, C(P[k]) from k = 1 to i
Else if (j = 0) return the sum of C(C(S[k], - ) from k = 1 to j
Else {
      Return maximum of the following three quantities:
      V(i-1, j-1) + C(S[i], P[j])
      V(i-1,j) + C(S[i], -)
      V(i, j-1) + C(-, P[j])
}
```

Fig 1

Now consider the sequence of recursive function calls when we call *V(2, 3)*:



The figure above shows the branching sequence of function calls, each function calling additional function, until it reaches the boundary conditions. As can be seen from the above figure, a straightforward recursive computation generates a lot of redundant calculations. I've highlighted the number of times *V(0, 1)* would have been computed in red. Roughly speaking, if I call the function with *V(n, m)*, then this will generate three new function calls, each of which creates another 3 new function call, etc. until we reach $i = 0$ or $j = 0$. If $n \sim m$, we will be carrying on $O(3^n)$ function calls and the computational complexity will be exponential in the length of the string to be aligned.

Tabular Computation

When we think about the problem of computing the recursive function, to compute any *V(i, j)*, we only need to know V*(i-1,j-1)*, *V(i-1, j)*, and *V(i,j-1)*. So, if we compute the function in a systematic manner, we only need to compute each *V(i, j)* once. For example, for *V(2, 3)*, we could first compute all *V(0,0), V(0,1), V(0,2), V(0,3), V(1,0), V(2,0)* since these functions don't depend on other functions. This is shown in the table below, where X marks the computations:

V(i,j)	0	1	2	3
0	X	X	X	X
1	X			
2	X			

Now to compute *V(1,1)*, we know all the necessary previous function values: *V(0,0), V(1,0),* and *V(0,1)*. So we can fill in the table:

V(i,j)	0	1	2	3
0	X	X	X	X
1	X	X		
2	X			

If we proceed row by row, left to right, each time we will have all the necessary previous values so we can fill in the table in *O(nm)* time where *n* and *m* are the lengths of the two strings. The technique of finding a global optimum by building up from partial solutions (solutions to the shorter strings in the alignment case) in a systematic manner to avoid computing redundant values is called **Dynamic Programming**. Here, "programming" refers not to coding but a technique for optimization. This is somewhat due to historical leftover terminology. Unfortunately, the term is in wide use so you will see terms like "**linear programming**", "**integer programming**", etc., all of which refer to optimization techniques. Next, we look at the above "tabular computation" in detail and also how to recover the optimal alignment itself from computing the optimal value of the objective function.

Tabular Computing

The key to systematically computing the objective function is to understand what is implied when we are filling in the [*i, j*] cell. From the above example, let's consider this stage of the table:

V(i,j)	0	1	2	3
0	X	X	X	X
1	X	X	X	X
2	X	X	**X**	

When we are filling in the [2,2] cell (marked in red), what we are doing is considering how to extend the partial solutions represented by the neighboring cells (diagonally left, top, and left). Suppose this is for the strings S and P and we will assume that the column indices are referring to positions of S and the row indices are referring to the positions of P. (Of course, we could reverse this and it would not make a difference as long as we keep it consistent.) The solution for cell [2, 2] can be extended from one of three possible directions. The simplest case is extending from the partial solution to [1,1]. The cell [1, 1] represents the score of the optimal alignment for the prefix S[1] and P[1]. By definition of an alignment, the alignment of S[1] and P[1] is "filled" in the sense that both the left and right end of the alignments should have nothing dangling. In fact, the optimal alignment of S[1] and P[1] will be one of the following three cases:

```
S[1] -          S[1]          - S[1]
 -  P[1]        P[1]        P[1] -
```

We are going to extend one of these alignments to the alignment of the prefixes S[1…2] and P[1…2]. We don't know which of the three cases is the optimal alignment for S[1] and P[1], but we don't care because the arrangement of the base pairs in the S[1] and P[1] alignment will not affect the score of the extension to S[1…2] and P[1…2]. To make the extension we have to add the symbols S[2] and P[2] to the existing optimal alignment. There are again three possible ways:

XXXS[2] - XXX - P[2]	XXXS[2] XXXP[2]	XXX - S[2] XXXP[2] -

I use the notation:

```
XXX
XXX
```

to denote the previous optimal alignment, since we don't know its exact form and all we care about is that it is a block of two strings. In general, the score models of alignments typically have the constraint that the cost of aligning two base pairs against each other is less than the cost of two positions that are aligned with a gap against a base pair. That is, we don't allow the 1st or the 3rd column cases—if this weren't the case, most alignments would simply interleave base pairs against gap characters, leading to nonsensical alignment. So, in effect, if we include this constraint, then the only possible extension of the previous alignment with S[2] and P[2] addition is:

```
XXXS[2]
XXXP[2]
```

There are two other previous alignments that can be extended to that of [2,2]. We can have an optimal alignment for [2,1] and extend to [2,2] or an optimal alignment for [1,2] and extend to [2,2]. These two alignment extension will have one of the following two forms:

```
YYYYS[2]
YYYY -

ZZZZ -
ZZZZP[2]
```

YYYY and ZZZZ represent the optimal alignment of the smaller strings.

Of these three possible ways of extending smaller optimal alignments to [2,2], one or more lead to the optimal score for [2,2]. We say "one or more" because we can have identical scores and, therefore, multiple optimal alignments. To compute which ones are optimal extensions we need to know the scores for the previous alignments and then add the cost/benefit of the extension to [2,2]. We choose the best score of the three possible scores as the score for [2,2].

Here is a computed example for the strings S = "AAA" P = "AT" with a cost matrix where each match is +1, each mismatch (substitution) is -1, and each gap is -2. It is useful to add the strings to the scoring table so we can keep track of what we are doing.

	-	A	A	A

–	0	-2	-4	-6
A	-2			
T	-4			

At the step, we fill in the first row and column. Note that each row and column of the table starts with a gap symbol. The boundary cells represent an alignment of the strings against this gap symbol—i.e., alignment to a null string. For example, going across the top row, the cells are [0,0], [0,1], [0,2], [0,3]. The [0,0] cell is just the starting condition. Then [0,1] cell represents the optimal alignment of S[1] against the null string. The [0, 2] cell represents the optimal alignment of S[1…2] against the null string, and so on. So if we were to show the optimal alignments for the top row, it would look like:

[0,0]	[0,1]	[0,2]	[0,3]
NULL	A –	AA --	AAA ---

Given the cost matrix, the values to fill in for the marginal case should be obvious. Now we consider how to fill in the value for the cell [1,1].

	–	A	A	A
–	0	-2	-4	-6
A	-2	(-4, -4, **+1**)		
T	-4			

In the above table, I listed three numbers (-4, -4, +1) corresponding to the cost of extending from top, left, diagonal, respectively. Let's go over these one by one. The alignment to the left has the score -2. Extending this to the right, implies adding the first "A" of the top string (the S string) to the existing alignment. So, the extended alignment is:

```
XA
X–
```

where again I used the X's to denote the block of previous alignment. The previous alignment score was -2 and the pair (A:–) has the score -2 according to the cost matrix, yielding a total of -4. By the same argument, extending the alignment from top, corresponds to

```
X–
XA
```

which also yields a total score of -4. Finally, extending from the diagonal implies:

```
XA
XA
```

yielding 0 (score of [0,0]) + 1 = +1 score. Obviously, the optimal value is +1, so we would have put the score of +1 for the [1,1] cell.

Here is the finished table where I list all three scores for each cell and mark the optimal value by red:

	–	A	A	A
–	0	-2	-4	-6
A	-2	(-4, -4, **+1**)	(-6,**-1**,**-1**)	(-8,**-3**,**-3**)
T	-4	(**-1**,-6,-3)	(-3,-3,**0**)	(-5,**-2**,**-2**)

The optimal score on the bottom right most cell, -2, is the optimal score value for the final alignment. As can be seen from the steps we've taken if we are aligning strings of length *n* and *m*, with the similarity objective function given in Eq 1, the computational time complexity to obtain the optimal score is O(*nm*).

All of the above shows us how the recursive scheme and its computation works. But, isn't there a different way to obtain an optimal alignment? Let's consider aligning S = "ATA" and P = "AA", using the scoring scheme, +1 for match, -1 for mismatch, and -2 for a gap. First convince yourself that the optimal alignment is:

```
ATA
A-A
```

This is a small enough problem that we can exhaustively try out the alignments and find out that the above alignment with the score 0 is the optimal alignment. We now consider trying to optimally align one letter at a time from each string, say one letter from S then one letter from P, finding the best choice going forward rather that considering all the paths to current alignment as we did above. We first have:

```
A
A
```

Then, we would do:

```
AT
A-
```

Then,

```
AT
AA
```

Because that is a better score than

```
AT-
A-A
```

Which would have the score -3. What if we did one letter from P then one letter from S? We would have the following:

```
A-
AA

AT
AA
```

What if we were allowed to examine one letter from each of the string simultaneously? Well, you can immediately see that we would still end up at

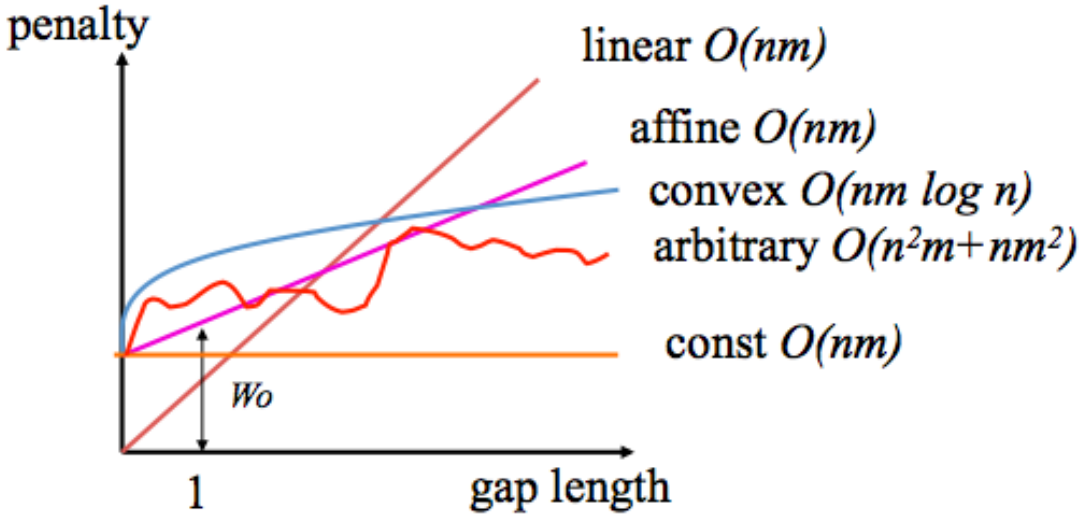
```
AT
AA
```

After these steps, we would add the final symbol to get the alignment:

```
ATA
AA-
```

with the score of -2.

The approach we have just taken is called a *"Greedy" approach*. We break the problem into smaller ones but at each step we greedly choose the best next step. There are certain types of problems where this approach is a good thing to do, but in the case of sequence alignment we see that the locally optimal step is not globally optimal; that is, the greedy approach gets us trapped in a local optimum. At each step, working off guaranteed best immediate solution and extending it forward does not work. Instead, we have to work backwards and ask, to get to X, what are the possible optimal paths to X? In the case of the similarity alignment in Eq 1, we were able to trace these optimal paths by considering all the possible one-step back--three of them. Why do we only need to look one step back? This has to do with the alignment model. The basic question for the alignment algorithm is whether it should introduce one or more gaps to align against one of the strings or whether it should introduce match/mismatches. When the cost of indels has a constant cost for each (thus, a linear function of length), this decision can be made one step at a time. Even in the case of the affine penalty function, we can still make the decision one step at a time because once we take into account whether a gap is a new one or an extension, the cost after that is constant per each indel. On the other hand, suppose we have a gap penalty function that is a variable function of the length, then considering one step is not enough since putting in enough gaps may increase the score rather than decreasing it. In which case, we have to keep track of all the possible optimal paths to X. The figure below shows the kinds of gap models we might consider and the computational time complexity of optimizing each kind of penalty function.



The main thing to remember is that each of these corresponds to a biological model. One way to think about the gap penalty models is as a heuristic probability model. Suppose *f(k)* is the unknown probability mass function for gaps of size *k* for evolutionary mutations. Then, the penalty function might be seen as a heuristic representation of this probability mass function, typically creating a cost value that is inversely proportional to the probability of the event. As mentioned above, an exact stochastic model incorporating a continuous time probability model has not be developed. The various gap penalty functions will correspond with a stochastic model if the penalty function reflects the **marginal distribution** of the stochastic model; that is, if it reflects the probability of seeing gaps of size *k* given that the stochastic process has run for some time period *t*. Unfortunately, for many biological reasonable models we do not have derivation of such marginal distributions.

Obtaining the optimal alignment

The recursive equations (e.g., Eq 1) and the tabular computation only return the score of the optimal alignment, not the optimal alignment itself. If we know the optimal score, is there a way to know the optimal alignment? Suppose we know that the optimal score for aligning S = "ATA" and P = "AA" is 0. Recall that this optimal score for S[1…3] and P[1…2] is an extension from one of the three possible partial alignments: S[1…3]-P[1], S[1…2]-P[1…2], and S[1…2]-P[1…2]. That is we know that the score is from one of these three cases:

V(3,1) + (-2) → (-:A)
V(2,2) + (-2) → (A:-)
V(2,1) + 1 → (A:A)

where I have indicated the implied added alignment on the right side of the arrow and the function V(*i,j*) returns the optimal score of aligning the strings S[1…*i*] and P[1…*j*]. We can compute the optimal score of any of the three partial alignments using the same procedure, so we assume we will always also know the value of V(*i-1,j*), V(*i, j-1*), and V(*i-1, j-1*). Suppose the optimal score of V(2,1) = -1 while V(2,2) = V(3,1) = 0. Then V(2,1) + 1 yields the KNOWN optimal score 0. The other two cases would have led to the suboptimal score of -2. So, by this argument we know the alignment of the last position has to be (A:A), and that amongst the three possible ways of getting the final optimal alignment, we built it from the optimal alignment of [2,1]. We can now repeat the procedure to find out the optimal alignment corresponding to V(2,1). Successively applying this procedure we will eventually get to V(0,0) and the whole optimal alignment. Thus, we can reconstruct the optimal alignment using only an algorithm to compute the score and not the alignment. For many optimization problems, if we know the optimal score, we can usually use a similar process to find the optimal configuration that led to the optimal score. However, as you can see this post-hoc reconstruction is an inefficient way to obtain the optimal alignment. We can easily retrieve the optimal alignment if we keep track of the choices we make during the computation of the optimal score. The following algorithm is called the "**traceback**" algorithm. Recall that when we are computing the optimal score during tabular computation:

V(i,j)	0	1	2	3
0	X	X	X	X
1	X	X	X	X
2	X	X	**X**	

We consider the score of the three possible extensions from the partial alignments:

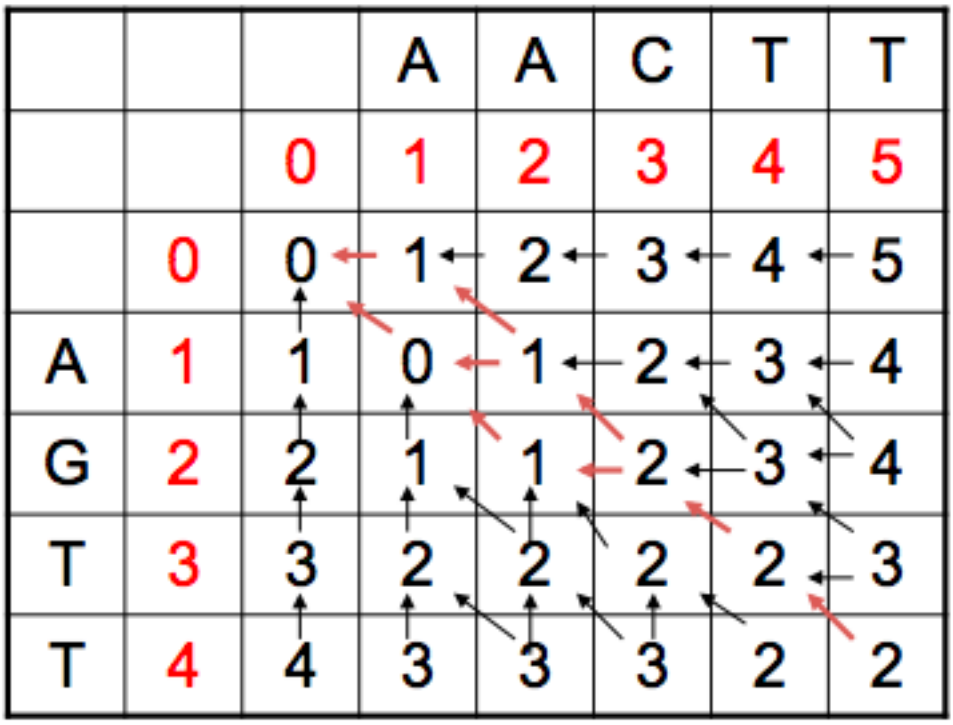
V(i,j)	0	1	2	3
0	X	X	X	X
1	X	X	X	X
2	X	X	**X**	

Suppose the optimal score for the (2, 2) cell (marked in red) was obtained from extending the score of the (1, 2) cell. Then, at the point of computation, we can keep a pointer to the partial alignment that yielded the particular score:

V(i,j)	0	1	2	3
0	X	X	X	X
1	X	X	X	X

2	X	X	**X**	

We can keep such pointers for each cell that we compute. Here is an example of an actual table:



Every cell has pointers (the arrows) but the main cell that is of interest to us is the bottom right most cell, which represents the score of the full alignment. We can start from the bottom-right most cell and trace the pointers back to the top-left most cell to figure out the optimal alignment. In this example, the pointer in the cell (4, 5) traces back to the cell (3, 4). This tells us the last position of the optimal alignment is (`T:T`). The cell (3, 4) traces back to cell (2, 3), resulting in the 2nd to last position optimal alignment of (`T:T`). From cell (2, 3), there are two trace back pointers. This can happen if there were two equally optimal choices. This tells us that there are at least two equally optimal alignments. The trace back to the cell (1, 2) implies the 3rd from last alignment of (`G:C`), while the trace back to the cell (2, 2) implies the 3rd from last alignment of (`C:-`). That is, we have the following optimal partial alignments up to this point:

```
...CTT
...GTT
```

and

```
...CTT
...-TT
```

The traceback from the cell (2, 2) points to (1, 1), which in turn points to (0, 0). This sequence results in the alignment:

```
AACTT
AG-TT
```

The traceback from the cell (1,2) splits into two possibilities again. There is a pointer to (1,1) and another to (0, 1), both of which in turns points to (0,0). The alignments for these two paths are:

```
AACTT
A-GTT

AACTT
-AGTT
```

So, the table above implies three total optimal alignments, all of which have the score 2.

The second variation for alignments involves modifying the scoring scheme to allow certain kinds of "free" variations. For example, suppose we want to allow the strings to have "free" gaps in the beginning or the end of the alignment. That is, allow alignments that look like this without any penalty for such hanging ends:

```
AAACCT---        ---AAACCT
----CTGAA        CTGAA-----
```

(Why would we want such alignments?) We can just change the boundary conditions above.

$V(0,j) = 0$
$V(i,0) = 0$ (Var 1)

The boundary conditions above, sets all the cells in the first row and the first column to zero. This is the same as saying that you can start the alignment with arbitrary number of gaps on one of the two strings; no matter how many starting gaps, the score is still zero. Such an alignment might be useful if we are trying to piece together short overlapping substrings from a long common string as might happen in shotgun sequencing.

A famous variation of a similar kind is the Smith-Waterman local alignment. In S-W local alignment, not only do we set the boundary conditions to zero, but we also change the recursion to:

$$V(i,j) = max\begin{cases} V(i-1,j-1) + C(S(i),P(j)) \\ V(i-1,j) + C(S(i),-) \\ V(i,j-1) + C(-,P(j)) \\ 0 \end{cases}$$ (Smith-Waterman)

Note the zero as an option. This means anywhere in the table, while computing an optimal score we can restart the alignment from the suffixes starting from that position (zero)—this is the very definition of a local alignment. It is the best possible alignment of any substring of the two original strings because the algorithm allows you to restart the alignment if the optimal score drops below zero. Of course, the scoring scheme has to be such that we expect positive values for good local alignments. Under the Smith-Waterman alignment scheme the optimal local alignment score can be found in any of the *n* by *m* cells, rather than the bottom-right most cell. This is because we are looking for any sub-alignments of substrings that might be locally best. When we do the traceback we also stop at when we reach a cell with value of zero. The length of the optimal local alignment is determined by the position of the cell(s) at which the top score is found and the cell at which a score of zero is found tracing back from the top scoring cell.