3 Exact matching

One of the most basic string operations is finding an exact copy of one string (often called query or pattern) within a longer string (text or reference). One can easily come up with simple examples of this operation in pretty much all text related applications, such as finding a word or short phrase within a webpage or document. Similarly, there are many such examples in biology. For example, restriction enzymes (proteins that 'cut' the DNA at specific locations in the genome) recognize exact patterns within a genome. Finding all such restriction sites is, thus, an instance of exact matching.

In this section we will try to develop fast algorithms for this seemingly simple problem. First, let's explore the following question:

Given a pattern (query), does it exist in the text (reference)?

Example:

```
text position     1  i           n = 17
Text              ATTCACTATTCGGCTAT
Pattern           GCAT
pattern position  1..m = 4
```

Before reading the following, try to come up with an algorithm to solve this problem. How 'expensive' is this algorithm?

As a quick aside – in Computer Science we generally compute the 'cost' of an algorithm in terms of two main commodities: time and memory needed to execute the algorithm. To simplify calculations at this point, assume that each object (letter, integer, etc.) uses up one memory location and that the only 'expensive' operation is the comparison of two characters. Thus, the run time of the algorithm would be expressed as the number of comparison operations. Also, as convention throughout the class, assume the text has length *n* and the pattern length *m*.

3.1 Naïve Approach

The most simple algorithm for exactly matching a string to another one works as follows. Start with the first character of the text and the first character of the pattern and compare each character individually. If a mismatch occurs shift the pattern one character and repeat the procedure. Terminate when either a match is found, or not enough characters remain in the text.

Here is the algorithm in pseudo-code:

```
for i=1,n - m + 1
     for j=1,m
          if P[j] != T[i+j-1]
               goto NOMATCH
      end for
     report match found @ position i
     NOMATCH
end for
```

Now, let us try to analyze this algorithm. First, its memory usage. Clearly we are only storing the pattern and the text (n + m memory locations) and a few additional variables (i, j, n, m), or a total of n + m + 4 memory locations. Since the memory usage is proportional to the total size of the inputs we say that our algorithm has 'linear memory usage'.

How about the run time? A quick back of the envelope calculation tells us that we repeat the comparison operation P[j] != T[i + j -1] m times in the inner for loop, which itself is repeated n – m + 1 times in the outer for loop, or a total of (n – m + 1) * m = nm – m^2 + m operations.

If we just want to coarsely compare the performance of different algorithms, we do not really need to go into this much detail, rather it is sufficient to focus on the largest term of the equation above and it suffices to say that our algorithm has a runtime 'order of' nm, also written as O(nm). This kind of reasoning is called 'asymptotic analysis' and focuses on just the behavior of the algorithm as the size of the inputs grows towards infinity. A full description of this approach is beyond the scope of this class and I encourage you to look up additional information if these terms are unfamiliar to you.

Question: *Does the algorithm above always perform $nm - m^2 + m$ operations?*

The answer depends on whether we stop as soon as we find a match, or we continue until we have found all the matches of the pattern within the text. In the latter case we will have to do all the comparisons to make sure we do not miss any matches. In the former, as soon as the first match is found the algorithm stops, possibly using a lot fewer operations.

Question: *Assume we just want to find the first match, do you ever need $nm - m^2 + m$ operations? If yes, when?*

The worst case scenario occurs when the pattern 'almost' matches the text, i.e., we have to compare characters all the way to the end of the pattern before figuring out the pattern doesn't match. See the example below.

```
1   i                n
AAAAAAAAAAAAAAAAA
AAAAT (m comparisons)
  AAAAT (m comparisons)
    AAAAT (m comparisons)
     ... (repeat for n-m+1 characters)
```

OK, so is the runtime the best we can do? Is it good enough for our typical applications? Let's imagine we want to match a 1000 bp sequence to the 3 billion bp human genome. The total runtime would be on the order of 3 10^{12} operations. On a 3 GHz processor (and assuming each comparison takes exactly one CPU cycle), the calculation would take 1,000 seconds, or about 16 minutes. Given that a typical sequencing experiment generates tens to hundreds of millions of sequences, this runtime is clearly too slow for even the simplest applications.

But how can we improve on this algorithm? The key idea is to use information we've already figured out when making earlier comparisons. If we can somehow reuse our work we might be able to save time. To illustrate this idea we will start with a simple algorithm, the Z algorithm, developed by Dan Gusfield [2] as an educational tool to help explain the basics of other string matching algorithms.

3.2 Z Algorithm

For a text T, let us define a simple function Z that captures the internal structure of the this text. Specifically, for every position $i \in (0, n]$ in T, we define Z[i] to be the length of the longest prefix of T[i..n] that matches exactly a prefix of T[0..n]. We also define Z[0] = 0. Here is an example:

Text:	A	T	T	C	A	C	T	A	T	T	C	G	G	C	T	A	T
Z[i]	0	0	0	0	1	0	0	4	0	0	0	0	0	0	0	2	0

Question: *Can Z values help in aligning a pattern to a text?*

To use Z values as a tool for exact matching, we can construct the following string S = P$T, where P and T are the pattern and text, respectively, and $ is a character that does not match any other character in either pattern or text. Let us assume for the moment that we have a black box function that computes the Z values for string S.

Question: *Can the Z values computed for the string S defined above help us find matches between P and T?*

The answer is yes – it suffices to look at the Z values corresponding to characters in T. Any Z-value equal to the length of P indicates the pattern matches. Specifically, for any i > m, if Z[i] == m, pattern P matches text T starting at position i. (**Aside:** can Z[i] be greater than m?)

Question: *What is the runtime and memory usage of this algorithm?*

First, the memory – we clearly need to store the Z values in addition to the pattern and text, for a total of *2(n + m) + 1* memory locations. Even though we are using more memory than the naïve algorithm, the memory consumption is still linear in the size of the inputs. There might be additional memory required by the algorithm that computes the Z values, but we'll get to that in a moment.

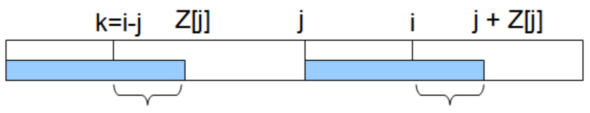
In terms of runtime, again omitting the work done to compute the Z values, we simply need to examine each position in the Z array to find all matches between P and T, or to decide that a match doesn't exist, i.e., the runtime is simply O(n) (we only need to examine the Z values corresponding to positions in the text). Here we see a much bigger improvement over the earlier naïve algorithm.

To sum up, if we have available an approach for computing Z values, we can perform exact matching much faster than the naïve algorithm, without using up (too much) more memory.

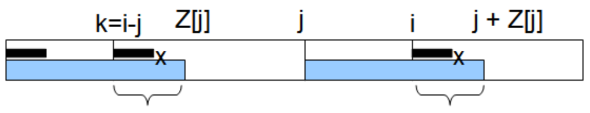
Of course, this result is predicated on the assumption that the Z values can be computed efficiently.

Question: *Can Z-values be computed in linear time and linear space?*

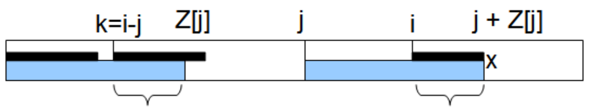
Note that one could come up with a naïve algorithm for computing the Z values using essentially the same approach used by the naïve exact matching algorithm. Clearly this approach would also yield a quadratic time algorithm (**Aside:** write down this algorithm and figure out its complexity in time and memory).



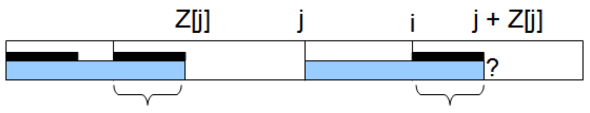
I.



II.



III.



A much faster algorithm can be developed if we cleverly reuse the work done earlier. The intuition is that when computing Z[i] we can use the Z[j] values for j < i. Let us look at the figure below, where we assume that there exists a position j < i such that the Z 'box' at position j extends past i, i.e., j + Z[j] > i (see the figure).

The section of the Z 'box' of j that extends past i is identical to the same region at the beginning of the string (bracket in the figure), information that we can now use to compute the Z[i] value.

We can distinguish several cases:

I. the Z 'box' for value k = i – j does not extend past the blue region (Z[j] box), k+Z[k] < Z[j]. In that case we know that Z[i] = Z[k] – the Z[i] box cannot extend past that point as the following character (marked with an x) is different from the corresponding character in the prefix of the string.

II. The Z 'box' for value k extends past the blue region (k + Z[k] > Z[j]). In this case Z[i] = j + Z[j] – i, i.e., the Z[i] box ends at the end of the Z[j] box. The reasoning is that the character marked with an x in the figure is different from the character at the end of the Z[j] box in the prefix of the string (otherwise Z[j] could have been extended), and thus the Z[i] box cannot be extended any further.

III. Finally, in the remaining case, where Z[k] + k = Z[j], we know that Z[i] >= j + Z[j] – i. The exact value cannot be known without checking the additional characters (starting with the one marked ? in the figure).

If you check carefully the three cases above, you can see that in cases I and II we can directly assign the Z[i] value without doing any additional comparisons, thus all the work is done in case III. We will show below that this approach allows us to dramatically speed up the algorithm.

But first, what j should we use? For any i, we will select the value j < i such that j + Z[j] > i and this value is maximal, i.e., the corresponding Z 'box' extends the farthest into the string. The intuition is that this way we are making maximal use of the information already computed. What happens, however, if no Z[j] box extends past i? In that case, we simply revert to the naïve algorithm, i.e. we compare T[i] to T[1], T[i + 1] to T[2], and so on and stop when we find a mismatch, appropriately updating the Z value.

Z algorithm:

```
Z[0] = 0
maxZ = 0
j = 0
for (i = 1; i < n; i++)
  if (maxZ < i)  // must do the hard work
    l = 0
    Z[i] = 0
    while (T[i + l] == T[l])
      Z[i]++
      l++
    end while
    maxZ = i + Z[i]
    j = i
  else
    k = i - j
    if (Z[k] + i < maxZ) // case 1
      Z[i] = Z[k]
    else if (Z[k] + i > maxZ) // case 2
      Z[i] = maxZ - i
    else // case 3 Z[k] + i = maxZ
      Z[i] = maxZ - i
      l = Z[i]
      while (T[i + l] == T[l])
        Z[i]++
        l++
      end while
      maxZ = i + Z[i]
      j = i
    end if
  end if
end for
```

Question: *Check the algorithm above and fix the 'off by one' errors (i.e., should k be $i - j$ or $i - j + 1$?)*

Now, why is this algorithm efficient? It should easy to see that other than a few additional variables we do not add any more memory, thus the memory usage remains linear. But how about the comparisons? They only occur within the two while loops, and only involve characters that have never been compared before, thus the total runtime cannot exceed the number of characters in the text, i.e., the runtime is linear.

Question: *Is it important that we only focus on the j whose Z[j] extends the furthest in the text? Argue/prove why this is important for ensuring either correctness or efficiency. (Hint: draw a situation where you are using a sub-optimal j, i.e., there exists another j you could use that extends further).*

Exercises

1. Implement the Z algorithm in your favorite programming language.

3.3 KMP – Knuth, Morris, Pratt Algorithm

As mentioned earlier, the Z algorithm was developed by Dan Gusfield as a way to teach string algorithms. The ideas we just presented should help guide you through the approach used by the Knuth Morris Pratt (KMP) string matching algorithm.

To develop the basic intuition about this algorithm, let us start with the naïve string matching algorithm described at the beginning of this chapter. The matching process compares the pattern to the text until either a match or a mismatch is found. In case of mismatch, the algorithm shifts the pattern position by one and restarts all the comparisons. Can we, however, reuse some of the information we learned while matching the pattern to the text. Let us look at the example below.

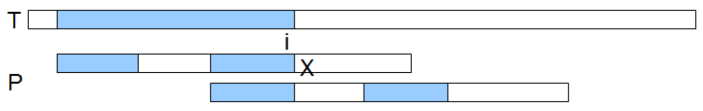
```
1 i              n
XYABCXABCXADCDAFEA
  ABCXABCDE
      ABCXABCDE
```

The naïve algorithm starts by matching all the characters highlighted in green, then mismatches on character D in the pattern, aligned to character X in the text. The intuition behind the KMP algorithm is that we can now simply shift the pattern over until its prefix matches a previously matched section of the text, as highlighted on the third row of the example. Thus, we can potentially save a lot of computation (comparisons) by not trying to match the pattern earlier.

Let us now formalize this approach better. We will define a new function, sp[i], which is defined for every position in the pattern. For every $i \in (1, m]$ sp[i] is the length of the longest non-trivial suffix of P[1..i] that matches exactly a prefix of the pattern.

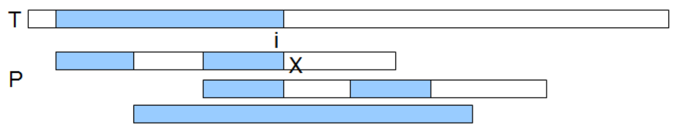
pattern:	A	T	T	C	A	C	T	A	T	T	C	G	G	C	T	A	T
sp[i]	0	0	0	0	1	0	0	1	2	3	4	0	0	0	0	1	2

As in the case of the Z algorithm we will assume for now that the sp[i] values can be computed efficiently. Can we use these values to speed up alignment? The basic idea is to shift the pattern i – sp[i] positions (rather than 1) once we mismatch at position i + 1, such that the prefix of the pattern is aligned to the corresponding matching suffix (see figure below). The matching continues from where it stopped (position in the text aligned to the X in the figure) as we already know the shaded region matches the text.



Question: *Are we missing any matches when shifting so far?*

To convince ourselves (i.e. prove) that the shifts are correct, i.e., we are not skipping any possible matches, we proceed with a proof by contradiction. Assume that there is another match between the original position of the pattern and the new one (which would be missed if we just jump along). The figure below shows this situation.



Assume the completely shaded pattern matches the text fully. As a corollary, the prefix of the pattern matches within the text in the same region where the pattern orginally matched (region before the X at the first position of the pattern), region that also matches the suffix of P[1..i] represented by the sp[i] value. This is clearly a contradiction with our initial assumption that sp[i] is the maximal such value. QED

Question: *Is the KMP algorithm efficient?*

Let us concentrate on the number of comparisons we make. It should be obvious that once a character in the text was matched with a character in the pattern, we never examine that specific character again. At the same time, the pattern can mismatch the same character in the text (that is aligned to the X in the figure) multiple times. Notice, however, that every time we hit a mismatch, we shift the pattern by at least one position (sp[i] < i), thus the number of shifts (and therefore comparisons) is bounded by the total length of the text. To sum it up, during our algorithm we encounter at most n correct matches, and we shift the pattern at most n times, for a total run time of 2n, i.e., the KMP algorithm is efficient. The memory usage is also good – we only use an additional m memory locations to store the sp[i] values.

The only bit remaining is computing the sp[i] values in linear time. The most simple approach is to use the Z algorithm, as there should be obvious similarities between the two concepts. Before you attempt to write such an algorithm, let us make a small change to the KMP algorithm. Let us define a new sp array, sp', where sp'[i] is the length of the longest suffix of P[1..i] that matches a prefix of P, **and** P[sp'[i] + 1] != P[i + 1] (the character after the matching suffix-prefix is different).

You can quickly check that running the KMP algorithm using the sp' values is equally correct and efficient. The new values are, however, easier to compute using the Z values.

Note: The sp' values highlight an interesting aspect – we are essentially being smarter about predicting what character is being aligned next (or more precisely what character isn't aligned next). An obvious extension of the sp' values would give us even more advance information. Since it's so obvious, we'll discuss it in class.

3.3.1 <u>Exercises</u>

1. Write out the algorithm for computing sp' values using the Z values.
2. Explain why the sp' values are easier to compute using Z values than the original sp values.
3. Implement, in your favorite language, the algorithms described above.
4. The genomes of many bacteria are circular. In order to represent the data in a linear form, genome assembly programs pick a random location in the genome to break the circle. Thus, it is possible that running the same program multiple times we would get different answers, corresponding to different circular rotations of the same string. Using the Z values described in class, describe a simple algorithm that identifies whether two DNA strings are circular rotations of each other. For example, string ACATTCAT is a circular rotation of string TTCATACA. (Hint: this algorithm is a simple extension of

the algorithm used to perform string matching based on Z values).

5. Write out the Z-values for the following string:

```
                    1 1 1 1 1 1 1 1 1 1 2 2 2 1
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
A G A G A G A C C A G C A G A G A G A G A C G T
```

Assume you are trying to compute Z[14] - please explain how you compute this value using the efficient algorithm described in class. Specifically:

a) what are the values for j and Z[j]?
b) which earlier value of Z do you need to make this computation?
c) do you need to perform any additional character comparisons?

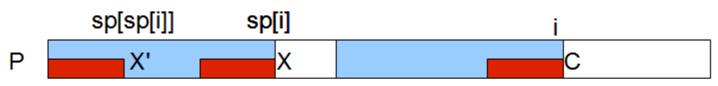
6. Give an example showing that k, the number of occurrences in T of patterns in set P, can grow faster than O(n + m). Be sure you account for the input size n. Try to make the growth as large as possible.
7. Define one possible relationship between sp(i) and sp(i – 1).
   Can you define a similar relationship between Z[i] and Z[i – 1]?

3.4 The original KMP computation of sp values

The Z algorithm and its use in the computation of the KMP sp' values has served its pedagogical role and has hopefully provided you with a better understanding of the way we reason about string matching. Below we will describe the original KMP algorithm (specifically the computation of the sp values). While we can clearly achieve the same results using the Z algorithm, the KMP approach will prove useful as we explore other aspects of string matching, such as the ability to simultaneously match multiple strings.

As you should already expect by now, the KMP algorithm achieves efficiency by reusing the results of earlier computations. This general optimization technique is called **memoization** and will pop up repeatedly in this class, most prominently when we discuss dynamic programming.

Thus, let us assume that we are trying to compute the value sp[i + 1] and that we already know the values sp[j] for all j <= i. Also, let us denote by c the character at position i + 1 in the pattern (c = P[i + 1]), and by x the character at position sp[i] + 1 (x = P[sp[i] + 1), i.e., the character that follows the prefix of P that matched the suffix ending at i. If the two characters match (x == c) we can simply extend the matching prefix/suffix pair by one character to obtain sp[i+1] = sp[i] + 1. What happens, however, if x != c? In this case sp[i] provides us with no additional information, but we might be able to use sp[sp[i]], the suffix of P[1..sp[i]] that matches a prefix of P. Here we will compare c with x', the character following this new prefix (see figure below). If they match, sp[i+1] = sp[sp[i]] + 1. If they don't, we simply continue trying further sp[sp[...[sp[i]]...]] values until we either find a match, or we 'bottom out' indicating no such prefix exists. In the latter case we simply set sp[i + 1] = 1 if c = P[1], or sp[i + 1] = 0, otherwise.



The correctness of this algorithm should be fairly obvious. Its efficiency, however, is unclear. At every position in the patter we might have to recurse through many sp[j] values before being able to compute the value of sp[i+1]. For now we will omit a proof of this property, which can be found in [2] pages 50-51[2]. The intuition behind the proof, however, is that we can 'charge' every recurrence to a previously matched character in the pattern. Thus, while any particular computation may 'stall' for a number of steps, the total number of recurrences will not exceed the length of the pattern, yielding an overall linear run time.