String Processing
Strings

What is string Matching? KMP,
One of the important (but relatively simple) algorithms for character strings is efficiently **searching for a substring or generally a pattern in large piece of text.**

Consider any finite alphabet $\Sigma$. A string $Y$ over $\Sigma$ is a finite sequence of symbols from $\Sigma$; $|Y|$ denotes length of $Y$, $Y$ is represented by an array $Y[0… m-1]$ where $m = |Y|$, $XY$ denotes concatenation 2 strings $X$ and $Y$ $[X.Y]$.

$Y[i:j]$, $0 \leq i \leq j$, denotes a **substring** of $Y$. $Y[0… i]$, $i \leq m$ is a **prefix** of $Y$; $Y[j:m]$, $j \leq m-1$, is a **suffix** of $Y$. Example: if $Y = “abcdf”, then, abc, a, bcd, cdf$ are all substrings of $Y$ while bdf, acd, cf are **not** substrings of $Y$.

A string $P$ [called **pattern**] occurs in $T$ [called **text**] at position $i$ [or, $P$ matches $T$ at position $i$], iff $P(j) = T(i+j–1)$, for $1 \leq j \leq |X|$

**Examples:** (1) Let $\Sigma = \{a, b, c\}$, and let $T = aabcabccaa$. Then, $P = abc$ is a substring of $T$ that occurs at positions 1 and 4. The prefix $T(0:4)$ is the substring aabca, and the suffix $T(4:9)$ is the substring abccaa. Note: If the pattern $P$ is a substring of the text $T$, then we just need to identify the start position of $P$ in $T$. **String matching is a fundamental problem in text editing.** [Think of “grep” when we perform search]
Another interesting problem is identifying the **Longest Common Subsequence (LCS)** of two given strings. Think of strings as sequence of characters. Given two strings $X = \{x_0, x_1, \ldots x_n\}$ and $Y = \{y_1, y_2, \ldots, y_m\}$, we say $Y$ is a subsequence of $X$ iff we can find a strictly increasing sequence of indices $\{i_1, i_2, \ldots, i_m \leq n\}$ such that $Y = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$.

If $X =$ (abracadabra) and $Y =$ (aadaa), $Y$ is a subsequence of $X$. Given two strings $X$ and $Y$, the longest common subsequence of $X$ and $Y$ is a longest sequence $Z$ which is both a subsequence of $X$ and $Y$. If $X =$ (abracadabra) and $Y =$ (yabbdabbdabdo), the longest common subsequence is (abadabda). It is not always unique – LCS of (abc) and (bac) is either (ac) or (bc).
Naïve String Matching

Here is the pseudocode: [Assume size of T is n and that of P is m]
for (i=0; T[i] != \0; i++) {
   // outer loop on T; i++ is executed no matter what happens in the inner loop
   for (j=0; T[i+j] != \0 && P[j] != \0 && T[i+j]==P[j]; j++) ;
   if (P[j] == \0) found a match}

Write the simple program and experiment with arbitrary strings. The inner one takes $O(m)$ iterations and the outer one takes $O(n)$ iterations so the total time is $O(mn)$. We want to do better. One worst case example: all characters in $T[]$ are "a"s, and $P[]$ is all "a"'s except for one "b" at the end. Then it takes $m$ comparisons each time to discover that you don't have a match, so $mn$ overall [This may occur in images and DNA sequences; unlikely in English text].

Note: We come out of the inner loop when $T[i+j] \neq P[j]$ [or, when either $T$ or $P$ is exhausted] and start with the next symbol in $T$ and compare it with symbols in $P$ starting at the beginning.

We need to do better. The idea is simple: while we are making progress through the loops, we are gathering more and more knowledge about the strings; we need to exploit that knowledge. We already have seen such an idea in converting the recursive function for Fibonacci numbers to a linear iterative function. This concept, in its generality, is known as Dynamic Programming ("Programming" in this context refers to a tabular method, not to writing computer code.) We have used this concept in solving the LCS problem.

We will use a slightly different approach to solve the string matching problem in linear time.
Notes: We come out of the inner loop when \( T[i+j] \neq P[j] \) [or, when either \( T \) or \( P \) is exhausted] and start with the next symbol in \( T \) and compare it with symbols in \( P \) starting at the beginning.

We need to do better. The idea is simple: while we are making progress through the loops, we are gathering more and more knowledge about the strings; we need to exploit that knowledge. We already have seen such an idea in converting the recursive function for Fibonacci numbers to a linear iterative function. This concept, in its generality, is known as Dynamic Programming ("Programming" in this context refers to a tabular method, not to writing computer code.)
A Typical Example

Suppose we're looking for pattern "nano" in text "banananobano". Each row represents an iteration of the outer loop, with each character in the row representing the result of a comparison (X if the comparison was unequal). Repeat the exercise with few other examples on your own.

Some of these comparisons are wasted work! For instance, (1) after iteration \(i=2\), we know from the comparisons we've done that \(T[3]="a"\), so there is no point comparing it to "n" in iteration \(i=3\).
(2) we also know that \(T[4]="n"\), so there is no point making the same comparison in iteration \(i=4\).

KMP algorithm idea is, in this sort of situation, after we've invested a lot of work making comparisons in the inner loop of the code, we know a lot about what's in the text. Specifically, if we've found a partial match of \(j\) characters starting at position \(i\), we know what's in positions \(T[i]...T[i+j-1]\). We need to make some more observations.
Observations

First, say the inner loop exits when \( T[i+j] \neq P[j] \); we know that \( P[j] \) is equal to elements in \( T[i] \ldots T[i+j-1] \). One can use this knowledge in two ways:

- One can skip some outer iterations, for which no match is possible; consider the rows
  
i=2: n a n
  
i=3: n a n o

Here the two placements of the pattern conflict with each other – we know from the \( i=2 \) iteration that \( T[3] \) and \( T[4] \) are "a" and "n", so they can't be the "n" and "a" that the \( i=3 \) iteration is looking for. We can keep skipping positions until we find one that doesn't conflict:

  i=2: n a n
  
i=4: n a n o

1. Here the two "n"s coincide. Define the overlap of two strings \( x \) and \( y \) to be the longest word that's a suffix of \( x \) and a prefix of \( y \). Here the overlap of "nan" and "nano" is just "n". (We don't allow the overlap to be all of \( x \) or \( y \), so it's not "nan").

2. In general, the value of \( i \) we want to skip to is the one corresponding to the largest overlap with the current partial match. Let’s write the code to skip the outer iterations; for the time being we assume the overlap function is provided (will design it soon).
String Matching with skipped outer iterations

```
i=0;
while (i<n)  {
for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;
if (P[j] == '\0') found a match;
i = i + max(1, j - overlap(P[0..j-1],P[0..m-1])); //we used the overlap function on P
}
```

We can also skip some inner iteration loops also. Look at the same example (we skipped from i=2 to i=4)

```
i=2: n a n
i=4: n a n o
```

In this example, the "n" that overlaps has already been tested by the i=2 iteration. There's no need to test it again in the i=4 iteration. In general, if we have a nontrivial overlap with the last partial match, we can avoid testing a number of characters equal to the length of the overlap. **Note:** the overlap function is a property of the P (pattern) array only; the T (text array) does not have anything to do with it. Now we can write the complete code for KMP search incorporating the skipping of inner iterations.

Computing the overlap function for a character array of length m can be done in O(m) time [we will see shortly] and is sometimes referred to as **preprocessing the pattern.**
**Complete KMP**

i = 0;
k = 0;
while (i<n) {
    for (j=k; T[i+j] != '' && P[j] != '' && T[i+j]==P[j]; j++) ;
    if (P[j] == '') found a match;
k = overlap(P[0..j-1], P[0..m]);
    // Note: k is never negative
    i = i + max(1, j-k); // i always advances by at least 1
}

- **We still have two loops;** is it possible that the time complexity is O(mn) in the worst case? NO. We will count the number of comparisons made by the algorithm.
- **Note that some comparisons return true while others return false.** If a comparison returns true, we've determined the value of T[i+j]. Then in future iterations, as long as there is a nontrivial overlap involving T[i+j], we'll skip past that overlap and not make a comparison with that position again. So, each position of T[] is only involved in one true comparison, and there can be n such comparisons total.
- On the other hand, there is at most one false comparison per iteration of the outer loop, so there can also only be n of those. As a result, we see that this part of the KMP algorithm makes at most 2n comparisons and takes time O(n).
- Only thing that remains is to compute the overlap function.

Note that we need to know overlap overlap(P[0..j-1], P) for all possible values of j, j > 0.
### Overlap function

Recall that **overlap of two strings x and y is the longest word that's a suffix of x and a prefix of y**; also, we need to know overlap overlap(P[0..j-1], P) for all possible values of j, j > 0. Once we've computed these values we can store them in an array and look them up when we need them. First, we introduce the design by using recursion.

We need to know for strings x and y not just the longest word that's a suffix of x and a prefix of y, but all such words. The key fact to notice here is that if w is a suffix of x and a prefix of y, and it's not the longest such word, then it's also a suffix of overlap(x,y). (This follows simply from the fact that it's a suffix of x that is shorter than overlap(x,y) itself.) So we can list all words that are suffixes of x and prefixes of y by the loop:

```plaintext
while (x != '\0') {x = overlap (x,y); output x;}
```

Now let's define **shorten(x)** to be the prefix of x with one fewer character. The next simple observation to make is that shorten(overlap(x,y)) is still a prefix of y, but is also a suffix of shorten(x). So, we can find overlap(x,y) by adding one more character to some word that's a suffix of shorten(x) and a prefix of y. We can just find all such words using the loop above, and return the first one for which adding one more character produces a valid overlap:
Overlap computation:

\[
z = \text{overlap}(\text{shorten}(x), y)
\]

while (last char of x != y[length(z)]) {
    if (z = empty) return overlap(x, y) = empty
    else z = overlap(z, y) 
}

return overlap(x, y);

You will need to work a little bit more to turn this into a working program.

Write an iterative function to store the overlap values in an array, to be used later.
### Examples for Overlap function

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>\0</td>
</tr>
<tr>
<td>overlap</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>\0</td>
</tr>
<tr>
<td>overlap</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>\0</td>
</tr>
<tr>
<td>overlap</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>\0</td>
</tr>
<tr>
<td>overlap</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
**Examples of KMP String Matching**

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|-------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Text  | A | B | A | B | D | A | B | A | C | D | A | B | A | B | C | A | B | A | B | \0 |
| Pattern| A | B | A | B | C | A | B | A | B | \0 |

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|-------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Text  | b | a | b | a | b | a | b | a | b | a | a | \0 |
| Pattern| a | b | a | b | \0 |

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|-------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Text  | c | c | g | c | t | a | c | c | g | c | g | a | t | a | c | \0 |
| Pattern| c | c | g | c | \0 |
Overlap Function of the Pattern

Overlap function is also known as prefix function; a slightly different way to do the same thing is known as failure function. In any case, this function is a property of the pattern; it has nothing to do with the text. Why?

Recall that overlap of two strings x and y is the longest word that's a suffix of x and a prefix of y; also, we need to know overlap overlap(P[0..j-1], P) for all possible values of j, 0 < j < m, where m is the length of the pattern P. Once we've computed these values, we can store them in an array and look them up when we need them.

It also indicates how much of the last comparison can be reused if it fails [outer loop]; it also helps to reduce the length of the inner loop in the next outer loop iteration.
void computeLPSArray(char *pat, int M, int *lps) {
    // *pat is the Pattern, M is length of pattern
    int len = 0;  // length of the previous longest prefix suffix
    int i; lps[0] = 0; // lps[0] is always 0; starting point
    i = 1;

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M) {
        if (pat[i] == pat[len])
            { len++; lps[i] = len; i++; }
        else { // (pat[i] != pat[len]) ⇒ mismatch
            if (len != 0) {
                // This is tricky. Consider the example
                // AAACAAAA and i = 7.
                len = lps[len-1]; // Why1?
                // Also, note that we do not increment i here Why2?
            }
            else // if (len == 0)
                { lps[i] = 0; i++; } }
    }
}

• LPS stands for longest prefix suffix; lps[] stores the overlap function of the Pattern P[]
• lps[i] = length of the longest word that's a suffix of P[0,j-1] and P[];
• Why1: since there’s a mismatch, we need to start from the previous character of the P array.
• Why2: Current character in the P array need now be compared again from the last match point.

Example 1

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>lps[0]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Example 2

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>lps[0]</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Complete KMP

```c
void KMPSearch(char *pat, char *txt) {
    int M = strlen(pat); int N = strlen(txt);
    // create lps[] that will hold the longest prefix suffix values for pattern
    int *lps = (int *)malloc(sizeof(int)*M);
    int j = 0;  // index for pat[]
    computeLPSArray(pat, M, lps); // Preprocess the pattern (calculate lps[] array)
    int i = 0;  // index for txt[]
    while (i < N)  { //
        if (pat[j] == txt[i])
            { j++; i++; }
        if (j == M)
            { printf("Found pattern at index %d \n", i-j); j = lps[j-1]; }  // mismatch after j matches
        else if (i < N && pat[j] != txt[i])  {
            // Do not match lps[0..lps[j-1]] characters, they will match anyway
            if (j != 0)
                j = lps[j-1];
            else
                i = i+1; }
    }
    free(lps); // to avoid memory leak }
```
Longest Common Subsequence (LCS)
**Longest Common Subsequence (LCS)**

We will first solve a much easier problem to get started. Consider two character arrays: \( P[m] \), \( T[n] \); we want to test if \( P[m] \) is a subsequence or not, assuming \( m < n \). [in applications, \( m \ll n \)]

```c
int main(void){
  char P[45] = "nematode knowledge"; char T[10] = "nano";
  while (*T != '\0')
    if (*P == (*T)++ && ++*P == '\0') {printf ("TRUE\n"); return(0);}
  printf ("FALSE\n"); return (0); }
```

We can visualize the process also as follows

Why do we want to solve the LCS problem?

- Molecular biology. DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four sub molecules forming DNA. When biologists find a new sequences, they typically want to know what other sequences it is most similar to.
- File comparison [Unix diff], Screen Redisplay by editors.
- Experiment with number of arbitrary strings to get the feel of what is happening.

This treatment is taken from David Eppstein of UCI
Some Simple Observations

Consider two arbitrary strings, one top of the other

| A | n | e | m | a | t | o | d | e | k | n | o | w | l | e | d | g | e |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | e | m | p | t | y | b | o | t | t | l | e |

Observe several important facts:

- If we draw the lines such that no line crosses any others, we get a common subsequence (not necessarily the longest though!); we can get many; which one is the longest?
- If the two strings A and B start with the same letter, it's always safe to choose that starting letter as the first character of the subsequence. Why?
- If the two first characters differ, then it is not possible for both of them to be part of a common subsequence – one or the other (or maybe both) will have to be removed.
- Once we've decided what to do with the first characters of the strings, the remaining subproblem is again a longest common subsequence problem, on two shorter strings. Therefore we can solve it recursively.
- It is much easier to compute the length of the LCS than the LCS itself by recursion; let's do that first.
Length of LCS by Recursion

// In the LCS problem, subproblems consist of a pair of suffixes of the two input strings.
int max (int a, int b)
{
if (a > b) return a;
else return b;
}
int lcs_length (char *A, char *B)
{
if (*A == '\0' || *B == '\0') return 0;
else if (*A == *B) return (1 + lcs_length(A+1, B+1));
else return max(lcs_length(A+1, B) , lcs_length(A, B+1));
}
int main()
{
char A[20] = "naematode knowledge";
char B[20] = "nanoee";
printf ("Length of LCS of A and B is %d\n", lcs_length (A, B));
return 0;
}

The worst running time is O(2^n) or exponential; for example, if the two strings have no matching characters, so the last line always gets executed.

Once we can make this recursive routine to an iterative routine, we will be able to recover the actual LCS. Our approach would be to use a 2-D array to store the subproblem results [the process is sometimes called memoization]. When we want the solution to a subproblem, we first look in the array, and check if there already is a solution there. If so we return it; otherwise we perform the computation and store the result. In the LCS problem, no result is negative, so we'll use -1 as a flag to tell the algorithm that nothing has been stored yet. It can be done in a relatively straightforward way; but there is a better way including this – called bottom – up dynamic programming.

We assume A and B contain valid strings. Can you rewrite to get LCS of any two subarrays of a given array? Try
It is similar to Fibonacci series; computing the same function again and again and over again.
Length of LCS using iteration

1. int lcs_length (char *A, char *B) {
2.   // Need a 2-D int array L,
3.   int i, j; int *L;
4.   int m = strlen(A), n = strlen(B);
5.   L = (int *) malloc ((m)*(n)*sizeof(int));
6.   for (i=m; i>= 0; i--)
7.     for (j=n; j>= 0; j--)
8.       {
9.         if (A[i] == '\0' || B[j]=='\0') L[i*m+j] = 0;
10.        else if (A[i] == B[j]) L[i*m+j] = 1 + L[(i+1)*m+j+1];
11.        else L[i*m+j] = max(L[(i+1)*m+j], L[i*m+j+1]);
12.     }
13.   return L[0]; }

1. The steps 1 – 5 allocates a 2-D array L of size #rows and #columns, which are respectively the lengths of the two strings – doesn’t matter which one is which. Everything else follows directly from the recursive formulation.

2. We think of the problem as a way of computing the entries in the array L where L[i,j] is the length of the length of LCS restricted to those substrings. We’d get the same results if we filled them in any order. We use something simpler, like a nested loop, that visits the array systematically. The only thing we must worry about is that when we fill in a cell L[i,j], we need to already know the values it depends on, namely in this case L[i+1,j], L[i,j+1], and L[i+1,j+1]. So, we traverse the array backwards, from the last row working up to the first and from the last column working up to the first. This is iterative (because it uses nested loops instead of recursion) or bottom up (because the order we fill in the array is from smaller simpler subproblems to bigger more complicated ones).

- Think of how 2-D arrays are stored in computer's memory.
- It’s easier, less error prone, to think of them as an array [read the matrix in a row major way] and adjust the index manipulations.
- Of course, it is possible to set it up so as to enable us to use the [][] notation for 2-D arrays; it’s a bit complicated
Retrieve the LCS itself

```c
int i = 0, j=0; char S[20]; int sp=0;
while (i<m && j<n) {
    if (A[i] == B[j]) {
        S[sp] = A[i]; sp++; i++; j++;
    } else if (L[(i+1)*m+j] >= L[i*m+j+1]) i++;
    else j++;
}
S[sp] = '\0'; // pad with a null to make it a string
```

Note:
- i and j are running indices on the character arrays A and B
- S is a character array to store the retrieved LCS and sp is an index to S, initialized at 0.
- One can print the LCS by using something like printf("\%s \n", S)
A Look at the array $L$

We ran the program for $A =$ “nematode knowledge“ and $B =$“empty bottle”. The LCS is “emt ole”
A Look at the array $L$

We ran the program for $A =$ “nematode knowledge“ and $B =$“empty bottle”. The LCS is “emt ole”
Pass a 2-D array to a function

```c
void func(int **array, int rows, int cols){
    int i, j;
    for (i=0; i<rows; i++)
    {
        for (j=0; j<cols; j++)
        {
            array[i][j] = i*j; //just an example
        }
    }
}

int main() {
    int rows, cols, i;
    int **x;
    /* obtain values for rows & cols */
    /* allocate the array */
    x = malloc(rows * sizeof *x);
    for (i=0; i<rows; i++)
    {
        x[i] = malloc(cols * sizeof *x[i]);
    }
    /* use the array */
    func(x, rows, cols);
    /* deallocate the array */
    for (i=0; i<rows; i++)
    { free(x[i]);}
    free(x);
}```